Sistemas Operacionais: Conceitos e Mecanismos

VIII - Aspectos de Segurança

Prof. Carlos Alberto Maziero
DAInf UTFPR
http://dainf.ct.utfpr.edu.br/~maziero

2 de junho de 2013

Este texto está licenciado sob a Licença *Attribution-NonCommercial-ShareAlike 3.0 Unported* da *Creative Commons* (CC). Em resumo, você deve creditar a obra da forma especificada pelo autor ou licenciante (mas não de maneira que sugira que estes concedem qualquer aval a você ou ao seu uso da obra). Você não pode usar esta obra para fins comerciais. Se você alterar, transformar ou criar com base nesta obra, você poderá distribuir a obra resultante apenas sob a mesma licença, ou sob uma licença similar à presente. Para ver uma cópia desta licença, visite http://creativecommons.org/licenses/by-nc-sa/3.0/.

Este texto foi produzido usando exclusivamente software livre: Sistema Operacional *GNU/Linux* (distribuições *Fedora* e *Ubuntu*), compilador de texto LaTeX 2ε, gerenciador de referências *BibTeX*, editor gráfico *Inkscape*, criadores de gráficos *GNUPlot* e *GraphViz* e processador PS/PDF *GhostScript*, entre outros.

Sumário

1 Introdução 4

2 Conceitos básicos 4
- 2.1 Propriedades e princípios de segurança . . . 5
- 2.2 Ameaças . . . 7
- 2.3 Vulnerabilidades . . . 7
- 2.4 Ataques . . . 9
- 2.5 Malwares . . . 11
- 2.6 Infraestrutura de segurança . . . 13

3 Fundamentos de criptografia 14
- 3.1 Cifragem e decifragem . . . 15
- 3.2 Criptografia simétrica e assimétrica . . . 15
- 3.3 Resumo criptográfico . . . 18
- 3.4 Assinatura digital . . . 19
- 3.5 Certificado de chave pública . . . 20

4 Autenticação 22
- 4.1 Usuários e grupos . . . 22
- 4.2 Técnicas de autenticação . . . 23
- 4.3 Senhas . . . 24
- 4.4 Senhas descartáveis . . . 25
- 4.5 Técnicas biométricas . . . 25
- 4.6 Desafio-resposta . . . 27
- 4.7 Certificados de autenticação . . . 28
- 4.8 Kerberos . . . 29
- 4.9 Infra-estruturas de autenticação . . . 31

5 Controle de acesso 33
- 5.1 Políticas, modelos e mecanismos de controle de acesso . . . 33
- 5.2 Políticas discricionárias . . . 35
  - 5.2.1 Matriz de controle de acesso . . . 35
  - 5.2.2 Tabela de autorizações . . . 37
  - 5.2.3 Listas de controle de acesso . . . 37
  - 5.2.4 Listas de capacidades . . . 39
- 5.3 Políticas obrigatórias . . . 40
  - 5.3.1 Modelo de Bell-LaPadula . . . 40
  - 5.3.2 Modelo de Biba . . . 41
  - 5.3.3 Categorias . . . 42
- 5.4 Políticas baseadas em domínios e tipos . . . 42
- 5.5 Políticas baseadas em papéis . . . 43
- 5.6 Mecanismos de controle de acesso . . . 45
  - 5.6.1 Infra-estrutura básica . . . 45
  - 5.6.2 Controle de acesso em UNIX . . . 46

5.6.3 Controle de acesso em Windows 48
5.6.4 Outros mecanismos 49
5.7 Mudança de privilégios 51

6 Auditoria **54**
6.1 Coleta de dados 54
6.2 Análise de dados 57
6.3 Auditoria preventiva 58

Resumo

Este módulo trata dos principais aspectos de segurança envolvidos na construção e utilização de um sistema operacional. Inicialmente são apresentados conceitos básicos de segurança e criptografia; em seguida, são descritos aspectos conceituais e mecanismos relacionados à autenticação de usuários, controle de acesso a recursos e serviços, integridade e privacidade do sistema operacional, das aplicações e dos dados armazenados. Grande parte dos tópicos de segurança apresentados neste capítulo não são exclusivos de sistemas operacionais, mas se aplicam a sistemas de computação em geral.

1 Introdução

A segurança de um sistema de computação diz respeito à garantia de algumas propriedades fundamentais associadas às informações e recursos presentes nesse sistema. Por "informação", compreende-se todos os recursos disponíveis no sistema, como registros de bancos de dados, arquivos, áreas de memória, portas de entrada/saída, conexões de rede, configurações, etc.

Em Português, a palavra "segurança" abrange muitos significados distintos e por vezes conflitantes. Em Inglês, as palavras "security", "safety" e "reliability" permitem definir mais precisamente os diversos aspectos da segurança: a palavra "security" se relaciona a ameaças intencionais, como intrusões, ataques e roubo de informações; a palavra "safety" se relaciona a problemas que possam ser causados pelo sistema aos seus usuários ou ao ambiente, como erros de programação que possam provocar acidentes; por fim, o termo "reliability" é usado para indicar sistemas confiáveis, construídos para tolerar erros de software, de hardware ou dos usuários [Avizienis et al., 2004]. Neste capítulo serão considerados somente os aspectos de segurança relacionados à palavra inglesa "security", ou seja, a proteção contra ameaças intencionais.

Este capítulo trata dos principais aspectos de segurança envolvidos na construção e operação de um sistema operacional. A primeira parte do capítulo apresentada conceitos básicos de segurança, como as propriedades e princípios de segurança, ameaças, vulnerabilidades e ataques típicos em sistemas operacionais, concluindo com uma descrição da infra-estrutura de segurança típica de um sistema operacional. A seguir, é apresentada uma introdução à criptografia. Na sequência, são descritos aspectos conceituais e mecanismos relacionados à autenticação de usuários, controle de acesso a recursos e integridade do sistema. Também são apresentados os principais conceitos relativos ao registro de dados de operação para fins de auditoria. Grande parte dos tópicos de segurança apresentados neste capítulo não são exclusivos de sistemas operacionais, mas se aplicam a sistemas de computação em geral.

2 Conceitos básicos

Nesta seção são apresentados alguns conceitos fundamentais, importantes para o estuda da segurança de sistemas computacionais. Em particular, são enumeradas as propriedades que caracterizam a segurança de um sistema, são definidos os principais

termos em uso na área, e são apresentados os principais elementos que compõe a arquitetura de segurança de um sistema.

2.1 Propriedades e princípios de segurança

A segurança de um sistema de computação pode ser expressa através de algumas propriedades fundamentais [Amoroso, 1994]:

Confidencialidade : os recursos presentes no sistema só podem ser consultados por usuários devidamente autorizados a isso;

Integridade : os recursos do sistema só podem ser modificados ou destruídos pelos usuários autorizados a efetuar tais operações;

Disponibilidade : os recursos devem estar disponíveis para os usuários que tiverem direito de usá-los, a qualquer momento.

Além destas, outras propriedades importantes estão geralmente associadas à segurança de um sistema:

Autenticidade : todas as entidades do sistema são autênticas ou genuínas; em outras palavras, os dados associados a essas entidades são verdadeiros e correspondem às informações do mundo real que elas representam, como as identidades dos usuários, a origem dos dados de um arquivo, etc.;

Irretratabilidade : Todas as ações realizadas no sistema são conhecidas e não podem ser escondidas ou negadas por seus autores; esta propriedade também é conhecida como *irrefutabilidade* ou *não-repudiação*.

É função do sistema operacional garantir a manutenção das propriedades de segurança para todos os recursos sob sua responsabilidade. Essas propriedades podem estar sujeitas a violações decorrentes de erros de software ou humanos, praticadas por indivíduos mal-intencionados (maliciosos), internos ou externos ao sistema.

Além das técnicas usuais de engenharia de software para a produção de sistemas corretos, a construção de sistemas computacionais seguros é pautada por uma série de princípios específicos, relativos tanto à construção do sistema quanto ao comportamento dos usuários e dos atacantes. Alguns dos princípios mais relevantes, compilados a partir de [Saltzer and Schroeder, 1975, Lichtenstein, 1997, Pfleeger and Pfleeger, 2006], são indicados a seguir:

Privilégio mínimo : todos os usuários e programas devem operar com o mínimo possível de privilégios ou permissões de acesso. Dessa forma, os danos provocados por erros ou ações maliciosas intencionais serão minimizados.

Mediação completa : todos os acessos a recursos, tanto diretos quanto indiretos, devem ser verificados pelos mecanismos de segurança. Eles devem estar dispostos de forma a ser impossível contorná-los.

Default seguro : o mecanismo de segurança deve identificar claramente os acessos permitidos; caso um certo acesso não seja explicitamente permitido, ele deve ser negado. Este princípio impede que acessos inicialmente não-previstos no projeto do sistema sejam inadvertidamente autorizados.

Economia de mecanismo : o projeto de um sistema de proteção deve ser pequeno e simples, para que possa ser facilmente e profundamente analisado, testado e validado.

Separação de privilégios : sistemas de proteção baseados em mais de um controle são mais robustos, pois se o atacante conseguir burlar um dos controles, mesmo assim não terá acesso ao recurso. Um exemplo típico é o uso de mais de uma forma de autenticação para acesso ao sistema (como um cartão e uma senha, nos sistemas bancários).

Compartilhamento mínimo : mecanismos compartilhados entre usuários são fontes potenciais de problemas de segurança, devido à possibilidade de fluxos de informação imprevistos entre usuários. Por isso, o uso de mecanismos compartilhados deve ser minimizado, sobretudo se envolver áreas de memória compartilhadas. Por exemplo, caso uma certa funcionalidade do sistema operacional possa ser implementada como chamada ao núcleo ou como função de biblioteca, deve-se preferir esta última forma, pois envolve menos compartilhamento.

Projeto aberto : a robustez do mecanismo de proteção não deve depender da ignorância dos atacantes; ao invés disso, o projeto deve ser público e aberto, dependendo somente do segredo de poucos itens, como listas de senhas ou chaves criptográficas. Um projeto aberto também torna possível a avaliação por terceiros independentes, provendo confirmação adicional da segurança do mecanismo.

Proteção adequada : cada recurso computacional deve ter um nível de proteção coerente com seu valor intrínseco. Por exemplo, o nível de proteção requerido em um servidor Web de serviços bancário é bem distinto daquele de um terminal público de acesso à Internet.

Facilidade de uso : o uso dos mecanismos de segurança deve ser fácil e intuitivo, caso contrário eles serão evitados pelos usuários.

Eficiência : os mecanismos de segurança devem ser eficientes no uso dos recursos computacionais, de forma a não afetar significativamente o desempenho do sistema ou as atividades de seus usuários.

Elo mais fraco : a segurança do sistema é limitada pela segurança de seu elemento mais vulnerável, seja ele o sistema operacional, as aplicações, a conexão de rede ou o próprio usuário.

Esses princípios devem pautar a construção, configuração e operação de qualquer sistema computacional com requisitos de segurança. A imensa maioria dos problemas de segurança dos sistemas atuais provém da não-observação desses princípios.

2.2 Ameaças

Como ameaça, pode ser considerada qualquer ação que coloque em risco as propriedades de segurança do sistema descritas na seção anterior. Alguns exemplos de ameaças às propriedades básicas de segurança seriam:

- *Ameaças à confidencialidade*: um processo vasculhar as áreas de memória de outros processos, arquivos de outros usuários, tráfego de rede nas interfaces locais ou áreas do núcleo do sistema, buscando dados sensíveis como números de cartão de crédito, senhas, e-mails privados, etc.;
- *Ameaças à integridade*: um processo alterar as senhas de outros usuários, instalar programas, *drivers* ou módulos de núcleo maliciosos, visando obter o controle do sistema, roubar informações ou impedir o acesso de outros usuários;
- *Ameaças à disponibilidade*: um usuário alocar para si todos os recursos do sistema, como a memória, o processador ou o espaço em disco, para impedir que outros usuários possam utilizá-lo.

Obviamente, para cada ameaça possível, devem existir estruturas no sistema operacional que impeçam sua ocorrência, como controles de acesso às áreas de memória e arquivos, quotas de uso de memória e processador, verificação de autenticidade de *drivers* e outros softwares, etc.

As ameaças podem ou não se concretizar, dependendo da existência e da correção dos mecanismos construídos para evitá-las ou impedi-las. As ameaças podem se tornar realidade à medida em que existam vulnerabilidades que permitam sua ocorrência.

2.3 Vulnerabilidades

Uma vulnerabilidade é um defeito ou problema presente na especificação, implementação, configuração ou operação de um software ou sistema, que possa ser explorado para violar as propriedades de segurança do mesmo. Alguns exemplos de vulnerabilidades são descritos a seguir:

- um erro de programação no serviço de compartilhamento de arquivos, que permita a usuários externos o acesso a outros arquivos do computador local, além daqueles compartilhados;
- uma conta de usuário sem senha, ou com uma senha pré-definida pelo fabricante, que permita a usuários não-autorizados acessar o sistema;
- ausência de quotas de disco, permitindo a um único usuário alocar todo o espaço em disco para si e assim impedir os demais usuários de usar o sistema.

A grande maioria das vulnerabilidades ocorre devido a erros de programação, como, por exemplo, não verificar a conformidade dos dados recebidos de um usuário ou da rede. Em um exemplo clássico, o processo servidor de impressão `lpd`, usado em alguns UNIX, pode ser instruído a imprimir um arquivo e a seguir apagá-lo, o que é útil para imprimir

arquivos temporários. Esse processo executa com permissões administrativas pois precisa acessar a porta de entrada/saída da impressora, o que lhe confere acesso a todos os arquivos do sistema. Por um erro de programação, uma versão antiga do processo `lpd` não verificava corretamente as permissões do usuário sobre o arquivo a imprimir; assim, um usuário malicioso podia pedir a impressão (e o apagamento) de arquivos do sistema. Em outro exemplo clássico, uma versão antiga do servidor HTTP Microsoft IIS não verificava adequadamente os pedidos dos clientes; por exemplo, um cliente que solicitasse a URL `http://www.servidor.com/../../../../windows/system.ini`, receberia como resultado o conteúdo do arquivo de sistema `system.ini`, ao invés de ter seu pedido recusado.

Uma classe especial de vulnerabilidades decorrentes de erros de programação são os chamados "estouros" de *buffer* e de pilha (*buffer/stack overflows*). Nesse erro, o programa escreve em áreas de memória indevidamente, com resultados imprevisíveis, como mostra o exemplo a seguir e o resultado de sua execução:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int i, j, buffer[20], k;  // declara buffer[0] a buffer[19]
5
6  int main()
7  {
8    i = j = k = 0 ;
9
10   for (i = 0; i<= 20; i++) // usa buffer[0] a buffer[20] <-- erro!
11      buffer[i] = random() ;
12
13   printf ("i: %d\nj: %d\nk: %d\n", i, j, k) ;
14
15   return(0);
16 }
```

A execução desse código gera o seguinte resultado:

```
1  host:~> cc buffer-overflow.c -o buffer-overflow
2  host:~> buffer-overflow
3  i: 21
4  j: 35005211
5  k: 0
```

Pode-se observar que os valores $i = 21$ e $k = 0$ são os previstos, mas o valor da variável j mudou "misteriosamente" de 0 para 35005211. Isso ocorreu porque, ao acessar a posição `buffer[20]`, o programa extrapolou o tamanho do vetor e escreveu na área de memória sucessiva[1], que pertence à variável j. Esse tipo de erro é muito frequente em linguagens como C e C++, que não verificam os limites de alocação das variáveis

[1]As variáveis não são alocadas na memória necessariamente na ordem em que são declaradas no código-fonte. A ordem de alocação das variáveis varia com o compilador usado e depende de vários fatores, como a arquitetura do processador, estratégias de otimização de código, etc.

durante a execução. O erro de estouro de pilha é similar a este, mas envolve variáveis alocadas na pilha usada para o controle de execução de funções.

Se a área de memória invadida pelo estouro de *buffer* contiver código executável, o processo pode ter erros de execução e ser abortado. A pior situação ocorre quando os dados a escrever no *buffer* são lidos do terminal ou recebidos através da rede: caso o atacante conheça a organização da memória do processo, ele pode escrever inserir instruções executáveis na área de memória invadida, mudando o comportamento do processo ou abortando-o. Caso o *buffer* esteja dentro do núcleo, o que ocorre em *drivers* e no suporte a protocolos de rede como o TCP/IP, um estouro de *buffer* pode travar o sistema ou permitir acessos indevidos a recursos. Um bom exemplo é o famoso *Ping of Death* [Pfleeger and Pfleeger, 2006], no qual um pacote de rede no protocolo ICMP, com um conteúdo específico, podia paralisar computadores na rede local.

Além dos estouros de *buffer* e pilha, há uma série de outros erros de programação e de configuração que podem constituir vulnerabilidades, como o uso descuidado das strings de formatação de operações de entrada/saída em linguagens como C e C++ e condições de disputa na manipulação de arquivos compartilhados. Uma explicação mais detalhada desses erros e de suas implicações pode ser encontrada em [Pfleeger and Pfleeger, 2006].

2.4 Ataques

Um ataque é o ato de utilizar (ou explorar) uma vulnerabilidade para violar uma propriedade de segurança do sistema. De acordo com [Pfleeger and Pfleeger, 2006], existem basicamente quatro tipos de ataques, representados na Figura 1:

Interrupção : consiste em impedir o fluxo normal das informações ou acessos; é um ataque à disponibilidade do sistema;

Interceptação : consiste em obter acesso indevido a um fluxo de informações, sem necessariamente modificá-las; é um ataque à confidencialidade;

Modificação : consiste em modificar de forma indevida informações ou partes do sistema, violando sua integridade;

Fabricação : consiste em produzir informações falsas ou introduzir módulos ou componentes maliciosos no sistema; é um ataque à autenticidade.

Existem ataques **passivos**, que visam capturar informações confidenciais, e ataques **ativos**, que visam introduzir modificações no sistema para beneficiar o atacante ou impedir seu uso pelos usuários válidos. Além disso, os ataques a um sistema operacional podem ser **locais**, quando executados por usuários válidos do sistema, ou **remotos**, quando são realizados através da rede, sem fazer uso de uma conta de usuário local. Um programa especialmente construído para explorar uma determinada vulnerabilidade de sistema e realizar um ataque é denominado *exploit*.

A maioria dos ataques a sistemas operacionais visa aumentar o poder do atacante dentro do sistema, o que é denominado *elevação de privilégios* (*privilege escalation*). Esses ataques geralmente exploram vulnerabilidades em programas do sistema (que executam

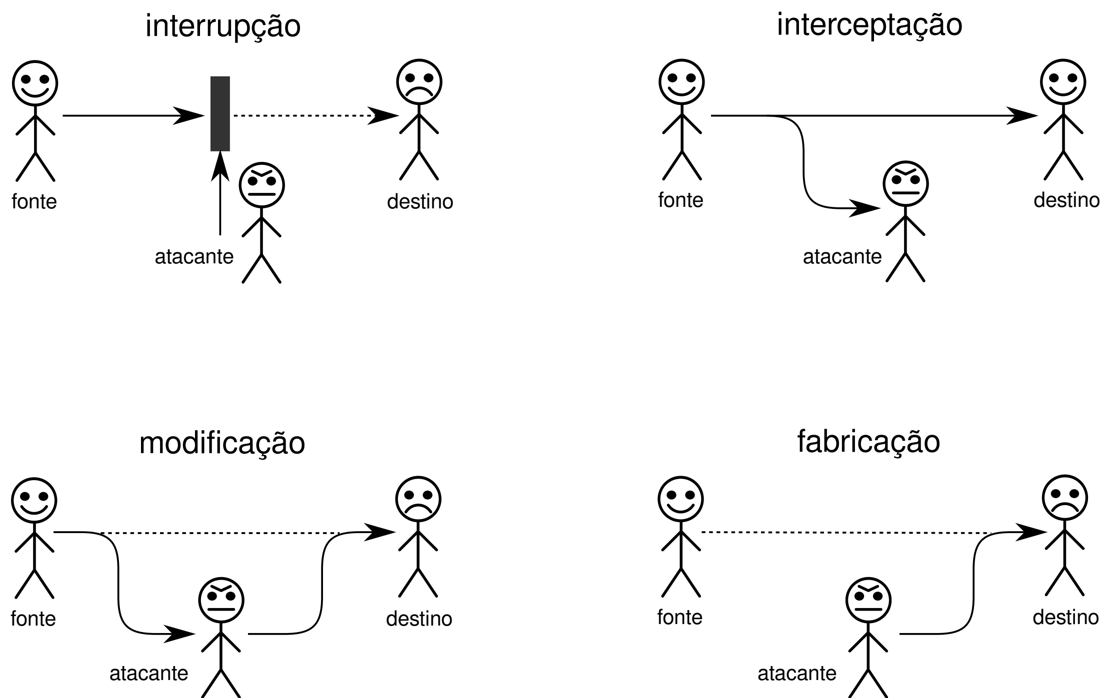

Figura 1: Tipos básicos de ataques (inspirado em [Pfleeger and Pfleeger, 2006]).

com mais privilégios), ou do próprio núcleo, através de chamadas de sistema, para receber os privilégios do administrador.

Por outro lado, os ataques de negação de serviços (DoS – *Denial of Service*) visam prejudicar a disponibilidade do sistema, impedindo que os usuários válidos do sistema possam utilizá-lo, ou seja, que o sistema execute suas funções. Esse tipo de ataque é muito comum em ambientes de rede, com a intenção de impedir o acesso a servidores Web, DNS e de e-mail. Em um sistema operacional, ataques de negação de serviço podem ser feitos com o objetivo de consumir todos os recursos locais, como processador, memória, arquivos abertos, *sockets* de rede ou semáforos, dificultando ou mesmo impedindo o uso desses recursos pelos demais usuários.

O antigo ataque *fork bomb* dos sistemas UNIX é um exemplo trivial de ataque DoS local: ao executar, o processo atacante se reproduz rapidamente, usando a chamada de sistema `fork` (vide código a seguir). Cada processo filho continua executando o mesmo código do processo pai, criando novos processos filhos, e assim sucessivamente. Em consequência, a tabela de processos do sistema é rapidamente preenchida, impedindo a criação de processos pelos demais usuários. Além disso, o grande número de processos solicitando chamadas de sistema mantém o núcleo ocupado, impedindo os a execução dos demais processos.

```
1  #include <unistd.h>
2
3  int main()
4  {
5    while (1)    // laço infinito
6       fork() ;  // reproduz o processo
7  }
```

Ataques similares ao *fork bomb* podem ser construídos para outros recursos do sistema operacional, como memória, descritores de arquivos abertos, *sockets* de rede e espaço em disco. Cabe ao sistema operacional impor limites máximos (quotas) de uso de recursos para cada usuário e definir mecanismos para detectar e conter processos excessivamente "gulosos".

Recentemente têm ganho atenção os ataques à confidencialidade, que visam roubar informações sigilosas dos usuários. Com o aumento do uso da Internet para operações financeiras, como acesso a sistemas bancários e serviços de compras *online*, o sistema operacional e os navegadores manipulam informações sensíveis, como números de cartões de crédito, senhas de acesso a contas bancárias e outras informações pessoais. Programas construídos com a finalidade específica de realizar esse tipo de ataque são denominados *spyware*.

Deve ficar clara a distinção entre *ataques* e *incidentes de segurança*. Um incidente de segurança é qualquer fato intencional ou acidental que comprometa uma das propriedades de segurança do sistema. A intrusão de um sistema ou um ataque de negação de serviços são considerados incidentes de segurança, assim como o vazamento acidental de informações confidenciais.

2.5 Malwares

Denomina-se genericamente *malware* todo programa cuja intenção é realizar atividades ilícitas, como realizar ataques, roubar informações ou dissimular a presença de intrusos em um sistema. Existe uma grande diversidade de *malwares*, destinados às mais diversas finalidades [Shirey, 2000, Pfleeger and Pfleeger, 2006], como:

Vírus : um vírus de computador é um trecho de código que se infiltra em programas executáveis existentes no sistema operacional, usando-os como suporte para sua execução e replicação[2]. Quando um programa "infectado" é executado, o vírus também se executa, infectando outros executáveis e eventualmente executando outras ações danosas. Alguns tipos de vírus são programados usando macros de aplicações complexas, como editores de texto, e usam os arquivos de dados dessas aplicações como suporte. Outros tipos de vírus usam o código de inicialização dos discos e outras mídias como suporte de execução.

Worm : ao contrário de um vírus, um "verme" é um programa autônomo, que se propaga sem infectar outros programas. A maioria dos vermes se propaga explorando vulnerabilidades nos serviços de rede, que os permitam invadir e instalar-se em sistemas remotos. Alguns vermes usam o sistema de e-mail como vetor de propagação, enquanto outros usam mecanismos de auto-execução de mídias removíveis (como *pendrives*) como mecanismo de propagação. Uma vez instalado em um sistema, o verme pode instalar *spywares* ou outros programas nocivos.

[2]De forma análoga, um vírus biológico precisa de uma célula hospedeira, pois usa o material celular como suporte para sua existência e replicação.

Trojan horse : de forma análoga ao personagem da mitologia grega, um "cavalo de Tróia" computacional é um programa com duas funcionalidades: uma funcionalidade lícita conhecida de seu usuário e outra ilícita, executada sem que o usuário a perceba. Muitos cavalos de Tróia são usados como vetores para a instalação de outros *malwares*. Um exemplo clássico é o famoso *Happy New Year 99*, distribuído através de e-mails, que usava uma animação de fogos de artifício como fachada para a propagação de um verme. Para convencer o usuário a executar o cavalo de Tróia podem ser usadas técnicas de *engenharia social* [Mitnick and Simon, 2002].

Exploit : é um programa escrito para explorar vulnerabilidades conhecidas, como prova de conceito ou como parte de um ataque. Os *exploits* podem estar incorporados a outros *malwares* (como vermes e *trojans*) ou constituírem ferramentas autônomas, usadas em ataques manuais.

Packet sniffer : um "farejador de pacotes" captura pacotes de rede do próprio computador ou da rede local, analisando-os em busca de informações sensíveis como senhas e dados bancários. A criptografia (Seção 3) resolve parcialmente esse problema, embora um *sniffer* na máquina local possa capturar os dados antes que sejam cifrados, ou depois de decifrados.

Keylogger : software dedicado a capturar e analisar as informações digitadas pelo usuário na máquina local, sem seu conhecimento. Essas informações podem ser transferidas a um computador remoto periodicamente ou em tempo real, através da rede.

Rootkit : é um conjunto de programas destinado a ocultar a presença de um intruso no sistema operacional. Como princípio de funcionamento, o *rootkit* modifica os mecanismos do sistema operacional que mostram os processos em execução, arquivos nos discos, portas e conexões de rede, etc., para ocultar o intruso. Os *rootkits* mais simples substituem utilitários do sistema, como `ps` (lista de processos), `ls` (arquivos), `netstat` (conexões de rede) e outros, por versões adulteradas que não mostrem os arquivos, processos e conexões de rede do intruso. Versões mais elaboradas de *rootkits* substituem bibliotecas do sistema operacional ou modificam partes do próprio núcleo, o que torna complexa sua detecção e remoção.

Backdoor : uma "porta dos fundos" é um programa que facilita a entrada posterior do atacante em um sistema já invadido. Geralmente a porta dos fundos é criada através um processo servidor de conexões remotas (usando SSH, telnet ou um protocolo ad-hoc). Muitos *backdoors* são instalados a partir de *trojans*, vermes ou *rootkits*.

Deve-se ter em mente que há na mídia e na literatura muita confusão em relação à nomenclatura de *malwares*; além disso, muitos *malwares* têm várias funcionalidades e se encaixam em mais de uma categoria. Esta seção teve como objetivo dar uma definição tecnicamente precisa de cada categoria, sem a preocupação de mapear os exemplos reais nessas categorias.

2.6 Infraestrutura de segurança

De forma genérica, o conjunto de todos os elementos de hardware e software considerados críticos para a segurança de um sistema são denominados **Base Computacional Confiável** (TCB – *Trusted Computing Base*) ou **núcleo de segurança** (*security kernel*). Fazem parte da TCB todos os elementos do sistema cuja falha possa representar um risco à sua segurança. Os elementos típicos de uma base de computação confiável incluem os mecanismos de proteção do hardware (tabelas de páginas/segmentos, modo usuário/núcleo do processador, instruções privilegiadas, etc.) e os diversos subsistemas do sistema operacional que visam garantir as propriedades básicas de segurança, como o controle de acesso aos arquivos, acesso às portas de rede, etc.

O sistema operacional emprega várias técnicas complementares para garantir a segurança de um sistema operacional. Essas técnicas estão classificadas nas seguintes grandes áreas:

Autenticação : conjunto de técnicas usadas para identificar inequivocamente usuários e recursos em um sistema; podem ir de simples pares *login/senha* até esquemas sofisticados de biometria ou certificados criptográficos. No processo básico de autenticação, um usuário externo se identifica para o sistema através de um procedimento de autenticação; no caso da autenticação ser bem sucedida, é aberta uma *sessão*, na qual são criados uma ou mais entidades (processos, *threads*, transações, etc.) para representar aquele usuário dentro do sistema.

Controle de acesso : técnicas usadas para definir quais ações são permitidas e quais são negadas no sistema; para cada usuário do sistema, devem ser definidas regras descrevendo as ações que este pode realizar no sistema, ou seja, que recursos este pode acessar e sob que condições. Normalmente, essas regras são definidas através de uma *política de controle de acesso*, que é imposta a todos os acessos que os usuários efetuam sobre os recursos do sistema.

Auditoria : técnicas usadas para manter um registro das atividades efetuadas no sistema, visando a contabilização de uso dos recursos, a análise posterior de situações de uso indevido ou a identificação de comportamento suspeitos.

A Figura 2 ilustra alguns dos conceitos vistos até agora. Nessa figura, as partes indicadas em cinza e os mecanismos utilizados para implementá-las constituem a base de computação confiável do sistema.

Sob uma ótica mais ampla, a base de computação confiável de um sistema informático compreende muitos fatores além do sistema operacional em si. A manutenção das propriedades de segurança depende do funcionamento correto de todos os elementos do sistema, do hardware ao usuário final.

O hardware fornece várias funcionalidades essenciais para a proteção do sistema: os mecanismos de memória virtual permitem isolar o núcleo e os processos entre si; o mecanismo de interrupção de software provê uma interface controlada de acesso ao núcleo; os níveis de execução do processador permitem restringir as instruções e as portas de entrada saída acessíveis aos diversos softwares que compõem o sistema; além

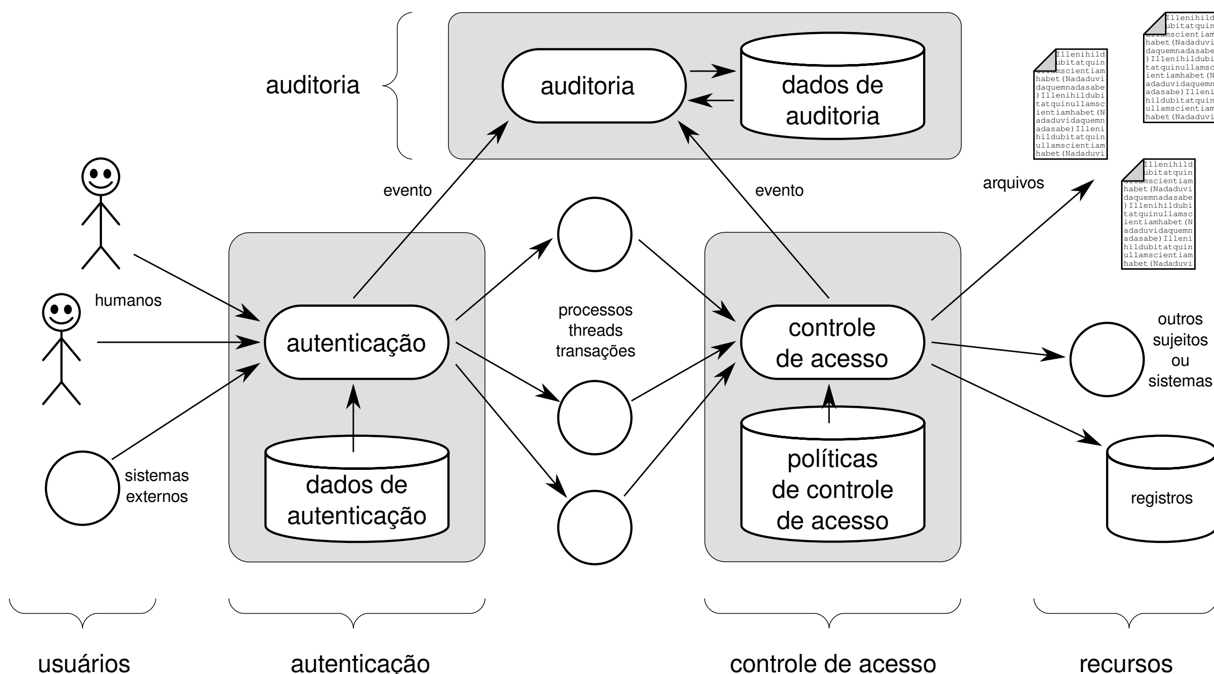

Figura 2: Base de computação confiável de um sistema operacional.

disso, muitos tipos de hardware permitem impedir operações de escrita ou execução de código em certas áreas de memória.

No nível do sistema operacional surgem os processos isolados entre si, as contas de usuários, os mecanismos de autenticação e controle de acesso e os registros de auditoria. Em paralelo com o sistema operacional estão os utilitários de segurança, como anti-vírus, verificadores de integridade, detectores de intrusão, entre outros.

As linguagens de programação também desempenham um papel importante nesse contexto, pois muitos problemas de segurança têm origem em erros de programação. O controle estrito de índices em vetores, a restrição do uso de ponteiros e a limitação de escopo de nomes para variáveis e funções são exemplos de aspectos importantes para a segurança de um programa. Por fim, as aplicações também têm responsabilidade em relação à segurança, no sentido de ter implementações corretas e validar todos os dados manipulados. Isso é particularmente importante em aplicações multi-usuários (como sistemas corporativos e sistemas Web) e processos privilegiados que recebam requisições de usuários ou da rede (servidores de impressão, de DNS, etc.).

3 Fundamentos de criptografia

As técnicas criptográficas são extensivamente usadas na segurança de sistemas, para garantir a confidencialidade e integridade dos dados. Além disso, elas desempenham um papel importante na autenticação de usuários e recursos. O termo "criptografia" provém das palavras gregas *kryptos* (oculto, secreto) e *graphos* (escrever). Assim, a criptografia foi criada para codificar informações, de forma que somente as pessoas autorizadas pudessem ter acesso ao seu conteúdo.

Alguns conceitos fundamentais para compreender as técnicas criptográficas são: o *texto aberto*, que é a mensagem ou informação a ocultar; o *texto cifrado*, que é a informação codificada; o *cifrador*, mecanismo responsável por cifrar/decifrar as informações, e as *chaves*, necessárias para poder cifrar ou decifrar as informações [Menezes et al., 1996].

3.1 Cifragem e decifragem

Uma das mais antigas técnicas criptográficas conhecidas é o *cifrador de César*, usado pelo imperador romano Júlio César para se comunicar com seus generais. O algoritmo usado nesse cifrador é bem simples: cada caractere do texto aberto é substituído pelo k-ésimo caractere sucessivo no alfabeto. Assim, considerando $k = 2$, a letra "A" seria substituída pela letra "C", a letra "R" pela "T", e assim por diante. Usando esse algoritmo, a mensagem secreta "Reunir todos os generais para o ataque" seria cifrada da seguinte forma:

mensagem aberta:	`Reunir todos os generais para o ataque`
mensagem cifrada com $k = 1$:	`Sfvojs upept pt hfofsbjt qbsb p bubrvf`
mensagem cifrada com $k = 2$:	`Tgwpkt vqfqu qu igpgtcku rctc q cvcswg`
mensagem cifrada com $k = 3$:	`Uhxqlu wrgrv rv jhqhudlv sdud r dwdtxh`

Para decifrar uma mensagem no cifrador de César, é necessário conhecer a mensagem cifrada e o valor de k utilizado para cifrar a mensagem, que é denominado *chave criptográfica*. Caso essa chave não seja conhecida, é possível tentar "quebrar" a mensagem cifrada testando todas as chaves possíveis, o que é conhecido como análise exaustiva ou de "força bruta". No caso do cifrador de César a análise exaustiva é trivial, pois há somente 26 valores possíveis para a chave k.

O número de chaves possíveis para um algoritmo de cifragem é conhecido como o seu *espaço de chaves*. De acordo com princípios enunciados pelo criptógrafo Auguste Kerckhoffs em 1883, o segredo de uma técnica criptográfica não deve residir no algoritmo em si, mas no espaço de chaves que ele provê. Seguindo esses princípios, a criptografia moderna se baseia em algoritmos públicos, bem avaliados pela comunidade científica, para os quais o espaço de chaves é muito grande, tornando inviável qualquer análise exaustiva. Por exemplo, o algoritmo de criptografia AES (*Advanced Encryption Standard*) adotado como padrão pelo governo americano, usando chaves de 128 bits, oferece um espaço de chaves com 2^{128} possibilidades, ou seja, 340.282.366.920.938.463.463.374.607.431.768.211.456 chaves diferentes... Se pudéssemos analisar um bilhão de chaves por segundo, ainda assim seriam necessários 10 sextilhões de anos para testar todas as chaves possíveis!

No restante do texto, a operação de cifragem de um conteúdo x usando uma chave k será representada por $\{x\}_k$ e a decifragem de um conteúdo x usando uma chave k será representada por $\{x\}_k^{-1}$.

3.2 Criptografia simétrica e assimétrica

De acordo com o tipo de chave utilizada, os algoritmos de criptografia se dividem em dois grandes grupos: *algoritmos simétricos* e *algoritmos assimétricos*. Nos **algoritmos**

simétricos, a mesma chave k usada para cifrar a informação deve ser usada para decifrá-la. Essa propriedade pode ser expressa em termos matemáticos:

$$\{ \{ x \}_k \}^{-1}_{k'} = x \iff k' = k$$

O cifrador de César é um exemplo típico de cifrador simétrico: se usarmos $k = 2$ para cifrar um texto, teremos de usar $k = 2$ para decifrá-lo. Os algoritmos simétricos mais conhecidos e usados atualmente são o DES (*Data Encryption Standard*) e o AES (*Advanced Encryption Standard*).

Os algoritmos simétricos são muito úteis para a cifragem de dados em um sistema local, como documentos ou arquivos em um disco rígido. Todavia, se a informação cifrada tiver de ser enviada a outro usuário, a chave criptográfica usada terá de ser informada a ele de alguma forma segura (de forma a preservar seu segredo). A Figura 3 ilustra o funcionamento básico de um sistema de criptografia simétrica.

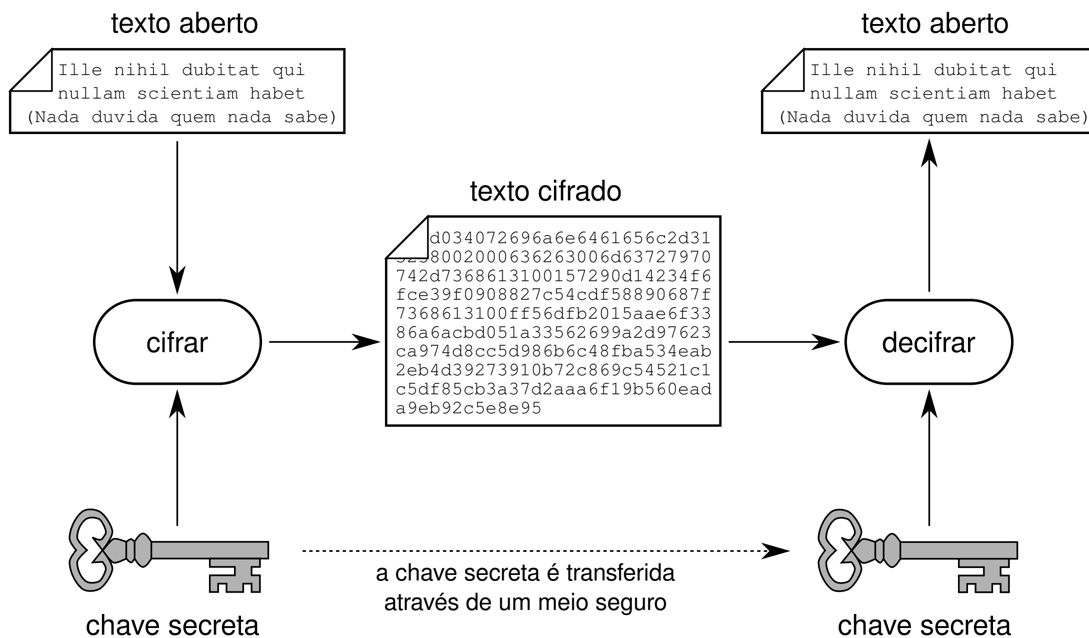

Figura 3: Criptografia simétrica.

Por outro lado, os **algoritmos assimétricos** se caracterizam pelo uso de um par de chaves complementares: uma *chave pública kp* e uma *chave privada kv*. Uma informação cifrada com uso de uma chave pública só poderá ser decifrada através da chave privada correspondente, e vice-versa. Considerando um usuário u com suas chaves pública $kp(u)$ e privada $kv(u)$, temos:

$$\begin{aligned} \{ \{ x \}_{kp(u)} \}^{-1}_k &= x \iff k = kv(u) \\ \{ \{ x \}_{kv(u)} \}^{-1}_k &= x \iff k = kp(u) \end{aligned}$$

Essas duas chaves estão fortemente relacionadas: para cada chave pública há uma única chave privada correspondente, e vice-versa. Todavia, não é possível calcular

uma das chaves a partir da outra. Como o próprio nome diz, geralmente as chaves públicas são amplamente conhecidas e divulgadas (por exemplo, em uma página Web ou um repositório de chaves públicas), enquanto as chaves privadas correspondentes são mantidas em segredo por seus proprietários. Alguns algoritmos assimétricos bem conhecidos são o RSA (*Rivest-Shamir-Adleman*) e o algoritmo de *Diffie-Hellman*. A Figura 4 ilustra o funcionamento da criptografia assimétrica.

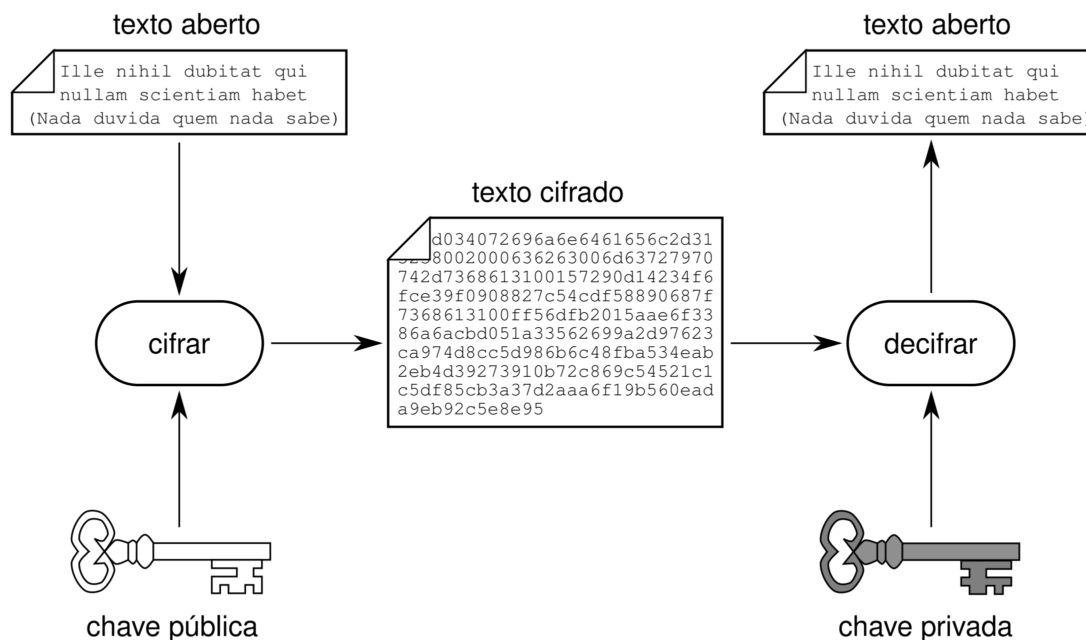

Figura 4: Criptografia assimétrica.

Um exemplo de uso da criptografia assimétrica é mostrado na Figura 5. Nele, a usuária Alice deseja enviar um documento cifrado ao usuário Beto[3]. Para tal, Alice busca a chave pública de Beto previamente divulgada em um chaveiro público (que pode ser um servidor Web, por exemplo) e a usa para cifrar o documento que será enviado a Beto. Somente Beto poderá decifrar esse documento, pois só ele possui a chave privada correspondente à chave pública usada para cifrá-lo. Outros usuários poderão até ter acesso ao documento cifrado, mas não conseguirão decifrá-lo.

A criptografia assimétrica também pode ser usada para identificar a autoria de um documento. Por exemplo, se Alice criar um documento e cifrá-lo com sua chave privada, qualquer usuário que tiver acesso ao documento poderá decifrá-lo e lê-lo, pois a chave pública de Alice está publicamente acessível. Todavia, o fato do documento poder ser decifrado usando a chave pública de Alice significa que ela é a autora legítima do mesmo, pois só ela teria acesso à chave privada que foi usada para cifrá-lo. Esse mecanismo é usado na criação das *assinaturas digitais* (Seção 3.4).

Embora mais versáteis, os algoritmos de cifragem assimétricos costumam exigir muito mais processamento que os algoritmos simétricos equivalentes. Por isso, muitas

[3]Textos em inglês habitualmente usam os nomes Alice, Bob, Carol e Dave para explicar algoritmos e protocolos criptográficos, em substituição às letras A, B, C e D. Neste texto usaremos a mesma abordagem, mas com nomes em português.

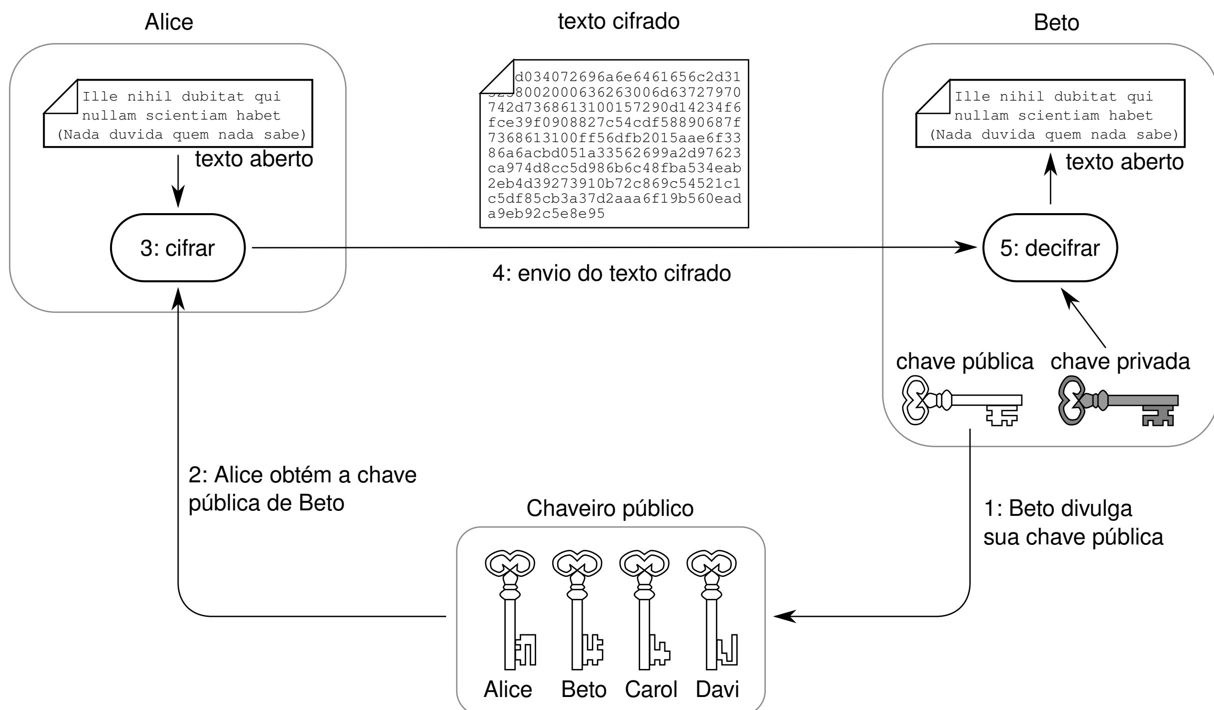

Figura 5: Exemplo de uso da criptografia assimétrica.

vezes ambos são usados em associação. Por exemplo, os protocolos de rede seguros baseados em TLS (*Transport Layer Security*), como o SSH e HTTPS, usam criptografia assimétrica somente durante o início de cada conexão, para negociar uma chave simétrica comum entre os dois computadores que se comunicam. Essa chave simétrica, chamada *chave de sessão*, é então usada para cifrar/decifrar os dados trocados entre os dois computadores durante aquela conexão, sendo descartada quando a sessão encerra.

3.3 Resumo criptográfico

Um *resumo criptográfico* (*cryptographic hash*) [Menezes et al., 1996] é uma função que gera uma sequência de bytes de tamanho pequeno e fixo (algumas dezenas ou centenas de bytes) a partir de um conjunto de dados de tamanho variável aplicado como entrada. Os resumos criptográficos são frequentemente usados para identificar unicamente um arquivo ou outra informação digital, ou para atestar sua integridade: caso o conteúdo de um documento digital seja modificado, seu resumo também será alterado.

Em termos matemáticos, os resumos criptográficos são um tipo de *função unidirecional* (*one-way function*). Uma função $f(x)$ é chamada unidirecional quando seu cálculo direto ($y = f(x)$) é simples, mas o cálculo de sua inversa ($x = f^{-1}(y)$) é impossível ou inviável em termos computacionais. Um exemplo clássico de função unidirecional é a fatoração do produto de dois números primos grandes: considere a função $f(p,q) = p \times q$, onde p e q são inteiros primos. Calcular $y = f(p,q)$ é simples e rápido, mesmo se p e q

forem grandes; entretanto, fatorizar y para obter de volta os primos p e q pode ser computacionalmente inviável, se y tiver muitos dígitos[4].

Idealmente, uma função de resumo criptográfico deve gerar sempre a mesma saída para a mesma entrada, e saídas diferentes para entradas diferentes. No entanto, como o número de bytes do resumo é pequeno, podem ocorrer *colisões*. Uma colisão ocorre quando duas entradas distintas x e x' geram o mesmo valor de resumo ($hash(x) = hash(x')$ para $x \neq x'$). Obviamente, bons algoritmos de resumo buscam minimizar essa possibilidade. Outras propriedades desejáveis dos resumos criptográficos são o *espalhamento*, em que uma modificação em um trecho específico dos dados de entrada gera modificações em partes diversas do resumo, e a *sensibilidade*, em que uma pequena modificação nos dados de entrada pode gerar grandes mudanças no resumo.

Os algoritmos de resumo criptográfico mais conhecidos e utilizados atualmente são o MD5 e o SHA1 [Menezes et al., 1996]. No Linux, os comandos `md5sum` e `sha1sum` permitem calcular respectivamente os resumos MD5 e SHA1 de arquivos comuns:

```
maziero:~> md5sum *
62ec3f9ff87f4409925a582120a40131  header.tex
0920785a312bd88668930f761de740bf  main.pdf
45acbba4b57317f3395c011fbd43d68d  main.tex
6c332adb037265a2019077e09a024d0c  main.tex~

maziero:~> sha1sum *
742c437692369ace4bf0661a8fe5741f03ecb31a  header.tex
9f9f52f48b75fd2f12fa297bdd5e1b13769a3139  main.pdf
d6973a71e5c30d0c05d762e9bc26bb073d377a0b  main.tex
cf1670f22910da3b9abf06821e44b4ad7efb5460  main.tex~
```

3.4 Assinatura digital

Os mecanismos de criptografia assimétrica e resumos criptográficos previamente apresentados permitem efetuar a *assinatura digital* de documentos eletrônicos. A assinatura digital é uma forma de verificar a autoria e integridade de um documento, sendo por isso o mecanismo básico utilizado na construção dos *certificados digitais*, amplamente empregados para a autenticação de servidores na Internet.

Em termos gerais, a assinatura digital de um documento é um resumo digital do mesmo, cifrado usando a chave privada de seu autor (ou de quem o está assinando). Sendo um documento d emitido pelo usuário u, sua assinatura digital $s(d, u)$ é definida por

$$s(d, u) = \{hash(d)\}_{kv(u)}$$

onde $hash(x)$ é uma função de resumo criptográfico conhecida, $\{x\}_k$ indica a cifragem de x usando uma chave k e $kv(u)$ é a chave privada do usuário u. Para verificar a validade da assinatura, basta calcular novamente o resumo $r' = hash(d)$ e compará-lo com o resumo obtido da assinatura, decifrada usando a chave pública de u ($r'' = \{s\}^{-1}_{kp(u)}$). Se ambos

[4]Em 2005, um grupo de pesquisadores alemães fatorizou um inteiro com 200 dígitos, usando 80 processadores Opteron calculando durante mais de de cinco meses.

forem iguais ($r' = r''$), o documento foi realmente assinado por u e está íntegro, ou seja, não foi modificado desde que u o assinou [Menezes et al., 1996].

A Figura 6 ilustra o processo de assinatura digital e verificação de um documento. Os passos do processo são:

1. Alice divulga sua chave pública kp_a em um repositório acessível publicamente;
2. Alice calcula o resumo digital r do documento d a ser assinado;
3. Alice cifra o resumo r usando sua chave privada kv_a, obtendo uma assinatura digital s;
4. A assinatura s e o documento original d, em conjunto, constituem o documento assinado por Alice: $[d, s]$;
5. Beto obtém o documento assinado por Alice ($[d', s']$, com $d' = d$ e $s' = s$ se ambos estiverem íntegros);
6. Beto recalcula o resumo digital $r' = hash(d')$ do documento, usando o mesmo algoritmo empregado por Alice;
7. Beto obtém a chave pública kp_a de Alice e a usa para decifrar a assinatura s' do documento, obtendo um resumo r'' ($r'' = r$ se s foi realmente cifrado com a chave kv_a e se $s' = s$) ;
8. Beto compara o resumo r' do documento com o resumo r'' obtido da assinatura digital; se ambos forem iguais ($r' = r''$), o documento foi assinado por Alice e está íntegro, assim como sua assinatura.

3.5 Certificado de chave pública

A identificação confiável do proprietário de uma chave pública é fundamental para o funcionamento correto das técnicas de criptografia assimétrica e de assinatura digital. Uma chave pública é composta por uma mera sequência de bytes que não permite a identificação direta de seu proprietário. Por isso, torna-se necessária uma estrutura complementar para fazer essa identificação. A associação entre chaves públicas e seus respectivos proprietários é realizada através dos *certificados digitais*. Um certificado digital é um documento digital assinado, composto das seguintes partes [Menezes et al., 1996]:

- A chave pública do proprietário do certificado;
- Identidade do proprietário do certificado (nome, endereço, e-mail, URL, número IP e/ou outras informações que permitam identificá-lo unicamente)[5];

[5]Deve-se ressaltar que um certificado pode pertencer a um usuário humano, a um sistema computacional ou qualquer módulo de software que precise ser identificado de forma inequívoca.

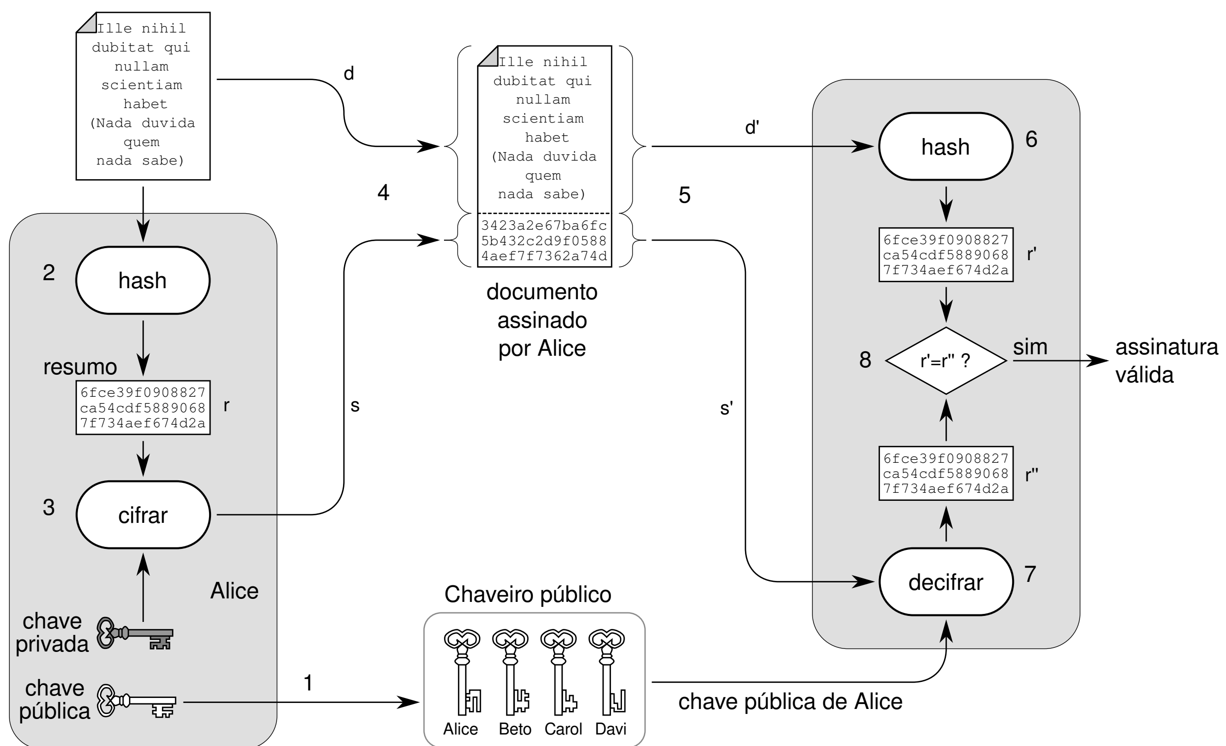

Figura 6: Assinatura e verificação de uma assinatura digital.

- Outras informações, como período de validade do certificado, algoritmos de criptografia e resumos preferidos ou suportados, etc.;
- Uma ou mais assinaturas digitais do conteúdo, emitidas por entidades consideradas confiáveis pelos usuários dos certificados.

Dessa forma, um certificado digital "amarra" uma identidade a uma chave pública. Para verificar a validade de um certificado, basta usar as chaves públicas das entidades que o assinaram. Existem vários tipos de certificados digitais com seus formatos e conteúdos próprios, sendo os certificados PGP e X.509 aqueles mais difundidos [Mollin, 2000].

Todo certificado deve ser assinado por alguma entidade considerada confiável pelos usuários do sistema. Essas entidades são normalmente denominadas *Autoridades Certificadoras* (AC ou CA – *Certification Authorities*). Como as chaves públicas das ACs devem ser usadas para verificar a validade de um certificado, surge um problema: como garantir que uma chave pública realmente pertence a uma dada autoridade certificadora? A solução é simples: basta criar um certificado para essa AC, assinado por outra AC ainda mais confiável. Dessa forma, pode-se construir uma estrutura hierárquica de certificação, na qual a AC de ordem mais elevada (denominada AC raiz) assina os certificados de outras ACs, e assim sucessivamente, até chegar aos certificados dos usuários e demais entidades do sistema. Uma estrutura de certificação se chama *Infra-estrutura de Chaves Públicas* (ICP ou PKI - *Public-Key Infrastructure*). Em uma ICP convencional (hierárquica), a chave pública da AC raiz deve ser conhecida de todos e é considerada íntegra [Mollin, 2000].

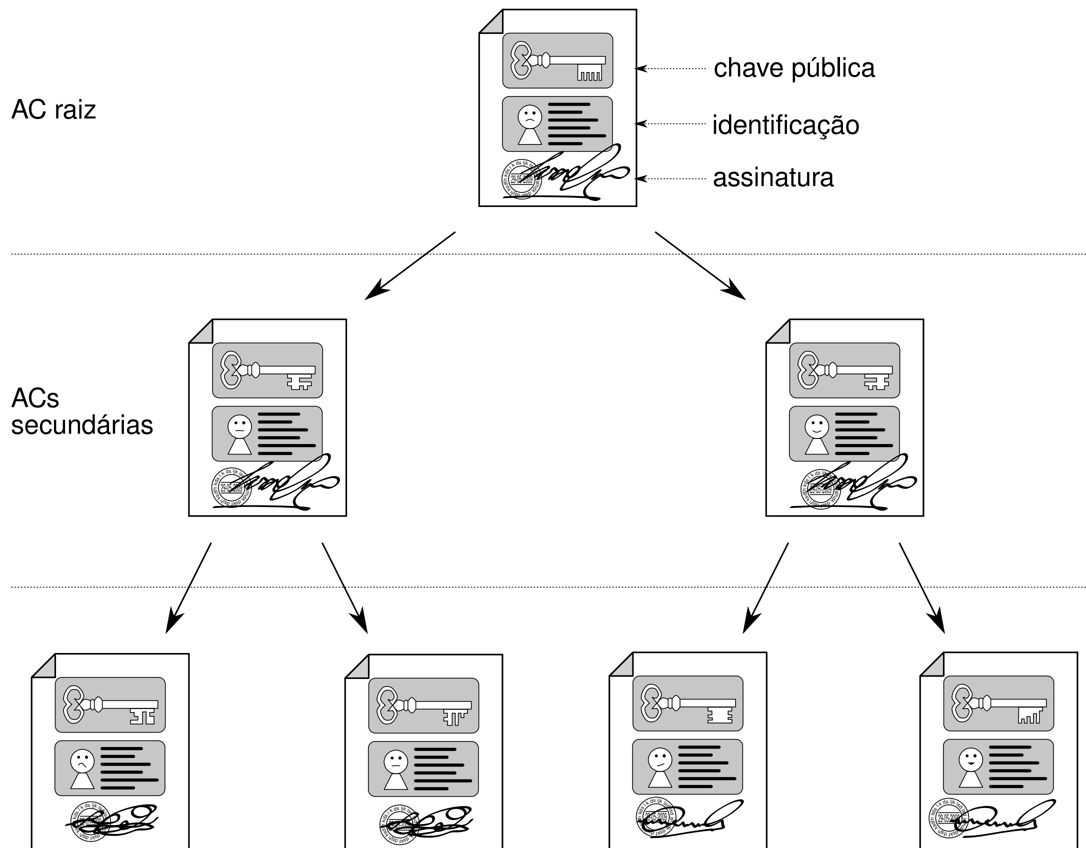

Figura 7: Infra-estrutura de chaves públicas hierárquica.

4 Autenticação

O objetivo da autenticação consiste em identificar as diversas entidades de um sistema computacional. Através da autenticação, o usuário interessado em acessar o sistema comprova que ele/a realmente é quem afirma ser. Para tal podem ser usadas várias técnicas, sendo as mais relevantes apresentadas nesta seção.

Inicialmente, a autenticação visava apenas identificar usuários, para garantir que somente usuários devidamente credenciados teriam acesso ao sistema. Atualmente, em muitas circunstâncias também é necessário o oposto, ou seja, identificar o sistema para o usuário, ou mesmo sistemas entre si. Por exemplo, quando um usuário acessa um serviço bancário via Internet, deseja ter certeza de que o sistema acessado é realmente aquele do banco desejado, e não um sistema falso, construído para roubar seus dados bancários. Outro exemplo ocorre durante a instalação de componentes de software como *drivers*: o sistema operacional deve assegurar-se que o software a ser instalado provêm de uma fonte confiável e não foi corrompido por algum conteúdo malicioso.

4.1 Usuários e grupos

A autenticação geralmente é o primeiro passo no acesso de um usuário a um sistema computacional. Caso a autenticação do usuário tenha sucesso, são criados processos

para representá-lo dentro do sistema. Esses processos interagem com o usuário através da interface e executam as ações desejadas por ele dentro do sistema, ou seja, agem em nome do usuário. A presença de um ou mais processos agindo em nome de um usuário dentro do sistema é denominada uma *sessão de usuário* (*user session* ou *working session*). A sessão de usuário inicia imediatamente após a autenticação do usuário (*login* ou *logon*) e termina quando seu último processo é encerrado, na desconexão (*logout* ou *logoff*). Um sistema operacional servidor ou *desktop* típico suporta várias sessões de usuários simultaneamente.

A fim de permitir a implementação das técnicas de controle de acesso e auditoria, cada processo deve ser associado a seu respectivo usuário através de um *identificador de usuário* (UID - *User IDentifier*), geralmente um número inteiro usado como chave em uma tabela de usuários cadastrados (como o arquivo `/etc/passwd` dos sistemas UNIX). O identificador de usuário é usado pelo sistema operacional para definir o proprietário de cada entidade e recurso conhecido: processo, arquivo, área de memória, semáforo, etc. É habitual também classificar os usuários em grupos, como *professores*, *alunos*, *contabilidade*, *engenharia*, etc. Cada grupo é identificado através de um *identificador de grupo* (GID - *Group IDentifier*). A organização dos grupos de usuários pode ser hierárquica ou arbitrária. O conjunto de informações que relaciona um processo ao seu usuário e grupo é geralmente denominado *credenciais do processo*.

Normalmente, somente usuários devidamente autenticados podem ter acesso aos recursos de um sistema. Todavia, alguns recursos podem estar disponíveis abertamente, como é o caso de pastas de arquivos públicas em rede e páginas em um servidor Web público. Nestes casos, assume-se a existência de um usuário fictício "convidado" (*guest*, *nobody*, *anonymous* ou outros), ao qual são associados todos os acessos externos não-autenticados e para o qual são definidas políticas de segurança específicas.

4.2 Técnicas de autenticação

As técnicas usadas para a autenticação de um usuário podem ser classificadas em três grandes grupos:

SYK – Something You Know ("algo que você sabe"): estas técnicas de autenticação são baseadas em informações conhecidas pelo usuário, como seu nome de *login* e sua senha. São consideradas técnicas de autenticação fracas, pois a informação necessária para a autenticação pode ser facilmente comunicada a outras pessoas, ou mesmo roubada.

SYH – Something You Have ("algo que você tem"): são técnicas que se baseiam na posse de alguma informação mais complexa, como um certificado digital ou uma chave criptográfica, ou algum dispositivo material, como um *smartcard*, um cartão magnético, um código de barras, etc. Embora sejam mais robustas que as técnicas SYK, estas técnicas também têm seus pontos fracos, pois dispositivos materiais, como cartões, também podem ser roubados ou copiados.

SYA – Something You Are ("algo que você é"): se baseiam em características intrinsecamente associadas ao usuário, como seus dados biométricos: impressão digital,

padrão da íris, timbre de voz, etc. São técnicas mais complexas de implementar, mas são potencialmente mais robustas que as anteriores.

Muitos sistemas implementam somente a autenticação por login/senha (SYK). Sistemas mais recentes têm suporte a técnicas SYH através de *smartcards* ou a técnicas SYA usando biometria, como os sensores de impressão digital. Alguns serviços de rede, como HTTP e SSH, também podem usar autenticação pelo endereço IP do cliente (SYA) ou através de certificados digitais (SYH).

Sistemas computacionais com fortes requisitos de segurança geralmente implementam mais de uma técnica de autenticação, o que é chamado de *autenticação multi-fator*. Por exemplo, um sistema militar pode exigir senha e reconhecimento de íris para o acesso de seus usuários, enquanto um sistema bancário pode exigir uma senha e o cartão emitido pelo banco. Essas técnicas também podem ser usadas de forma gradativa: uma autenticação básica é solicitada para o usuário acessar o sistema e executar serviços simples (como consultar o saldo de uma conta bancária); se ele solicitar ações consideradas críticas (como fazer transferências de dinheiro para outras contas), o sistema pode exigir mais uma autenticação, usando outra técnica.

4.3 Senhas

A grande maioria dos sistemas operacionais de propósito geral implementam a técnica de autenticação SYK baseada em *login/senha*. Na autenticação por senha, o usuário informa ao sistema seu identificador de usuário (nome de *login*) e sua senha, que normalmente é uma sequência de caracteres memorizada por ele. O sistema então compara a senha informada pelo usuário com a senha previamente registrada para ele: se ambas forem iguais, o acesso é consentido.

A autenticação por senha é simples mas muito frágil, pois implica no armazenamento das senhas "em aberto" no sistema, em um arquivo ou base de dados. Caso o arquivo ou base seja exposto devido a algum erro ou descuido, as senhas dos usuários estarão visíveis. Para evitar o risco de exposição indevida das senhas, são usadas funções unidirecionais para armazená-las, como os resumos criptográficos (Seção 3.3).

A autenticação por senhas usando um resumo criptográfico é bem simples: ao registrar a senha s de um novo usuário, o sistema calcula seu resumo ($r = hash(s)$), e o armazena. Mais tarde, quando esse usuário solicitar sua autenticação, ele informará uma senha s'; o sistema então calculará novamente seu resumo $r' = hash(s')$ e irá compará-lo ao resumo previamente armazenado ($r' = r$). Se ambos forem iguais, a senha informada pelo usuário é considerada autêntica e o acesso do usuário ao sistema é permitido. Com essa estratégia, as senhas não precisam ser armazenadas em aberto no sistema, aumentando sua segurança.

Caso um intruso tenha acesso aos resumos das senhas dos usuários, ele não conseguirá calcular de volta as senhas originais (pois o resumo foi calculado por uma função unidirecional), mas pode tentar obter as senhas indiretamente, através do *ataque do dicionário*. Nesse ataque, o invasor usa o algoritmo de resumo para cifrar palavras conhecidas ou combinações delas, comparando os resumo obtidos com aqueles presentes no arquivo de senhas. Caso detecte algum resumo coincidente, terá encontrado a senha correspondente. O ataque do dicionário permite encontrar senhas consideradas "fracas",

por serem muito curtas ou baseadas em palavras conhecidas. Por isso, muitos sistemas operacionais definem políticas rígidas para as senhas, impedindo o registro de senhas óbvias ou muito curtas e restringindo o acesso ao repositório dos resumos de senhas.

4.4 Senhas descartáveis

Um problema importante relacionado à autenticação por senhas reside no risco de roubo da senhas. Por ser uma informação estática, caso uma senha seja roubada, o malfeitor poderá usá-la enquanto o roubo não for percebido e a senha substituída. Para evitar esse problema, são propostas técnicas de senhas descartáveis (OTP - *One-Time Passwords*). Como o nome diz, uma senha descartável só pode ser usada uma única vez, perdendo sua validade após esse uso. O usuário deve então ter em mãos uma lista de senhas pré-definidas, ou uma forma de gerá-las quando necessário. Há várias formas de se produzir e usar senhas descartáveis, entre elas:

- Armazenar uma lista sequencial de senhas (ou seus resumos) no sistema e fornecer essa lista ao usuário, em papel ou outro suporte. Quando uma senha for usada com sucesso, o usuário e o sistema a eliminam de suas respectivas listas.

- Uma variante da lista de senhas é conhecida como *algoritmo OTP de Lamport* [Menezes et al., 1996]. Ele consiste em criar uma sequência de senhas $s_0, s_1, s_2, \cdots, s_n$ com s_0 aleatório e $s_i = hash(s_{i-1})\ \forall i > 0$, sendo $hash(x)$ uma função de resumo criptográfico conhecida. O valor de s_n é informado ao servidor previamente. Ao acessar o servidor, o cliente informa o valor de s_{n-1}. O servidor pode então comparar $hash(s_{n-1})$ com o valor de s_n previamente informado: se forem iguais, o cliente está autenticado e ambos podem descartar s_n. O servidor armazena s_{n-1} para validar a próxima autenticação, e assim sucessivamente. Um intruso que conseguir capturar uma senha s_i não poderá usá-la mais tarde, pois não conseguirá calcular $s_{i-1} = hash^{-1}(s_i)$.

- Gerar senhas temporárias sob demanda, através de um dispositivo ou software externo usado pelo cliente; as senhas temporárias podem ser geradas por um algoritmo de resumo que combine uma senha pré-definida com a data/horário corrente. Dessa forma, cliente e servidor podem calcular a senha temporária de forma independente. Como o tempo é uma informação importante nesta técnica, o dispositivo ou software gerador de senhas do cliente deve estar sincronizado com o relógio do servidor. Dispositivos OTP como o mostrado na Figura 8 são frequentemente usados em sistemas de *Internet Banking*.

4.5 Técnicas biométricas

A biometria (*biometrics*) consiste em usar características físicas ou comportamentais de um indivíduo, como suas impressões digitais ou seu timbre de voz, para identificá-lo unicamente perante o sistema. Diversas características podem ser usadas para a autenticação biométrica; no entanto, elas devem obedecer a um conjunto de princípios básicos [Jain et al., 2004]:

Figura 8: Um dispositivo gerador de senhas descartáveis.

- *Universalidade*: a característica biométrica deve estar presente em todos os indivíduos que possam vir a ser autenticados;
- *Singularidade* (ou unicidade): dois indivíduos quaisquer devem apresentar valores distintos para a característica em questão;
- *Permanência*: a característica não deve mudar ao longo do tempo, ou ao menos não deve mudar de forma abrupta;
- *Mensurabilidade*: a característica em questão deve ser facilmente mensurável em termos quantitativos.

As características biométricas usadas em autenticação podem ser *físicas* ou *comportamentais*. Como características físicas são consideradas, por exemplo, o DNA, a geometria das mãos, do rosto ou das orelhas, impressões digitais, o padrão da íris (padrões na parte colorida do olho) ou da retina (padrões de vasos sanguíneos no fundo do olho). Como características comportamentais são consideradas a assinatura, o padrão de voz e a dinâmica de digitação (intervalos de tempo entre teclas digitadas), por exemplo.

Os sistemas mais populares de autenticação biométrica atualmente são os baseados em impressões digitais e no padrão de íris. Esses sistemas são considerados confiáveis, por apresentarem taxas de erro relativamente baixas, custo de implantação/operação baixo e facilidade de coleta dos dados biométricos. A Figura 9 apresenta alguns exemplos de características biométricas empregadas nos sistemas atuais.

Um sistema biométrico típico é composto de um *sensor*, responsável por capturar dados biométricos de uma pessoa; um *extrator de características*, que processa os dados do sensor para extrair suas características mais relevantes; um *comparador*, cuja função é comparar as características extraídas do indivíduo sob análise com dados previamente armazenados, e um *banco de dados* contendo as características biométricas dos usuários registrados no sistema [Jain et al., 2004]. O sistema pode funcionar de dois modos: no modo de *autenticação*, ele verifica se as características biométricas de um indivíduo (previamente identificado por algum outro método, como login/senha, cartão, etc.) correspondem às suas características biométricas previamente armazenadas. Desta forma, a biometria funciona como uma autenticação complementar. No modo de

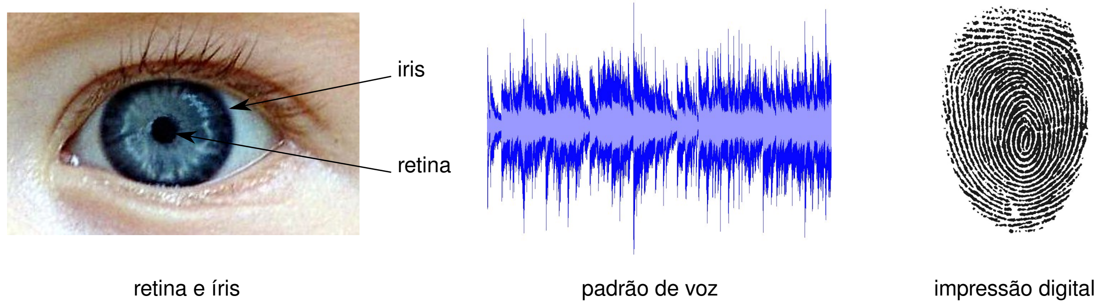

Figura 9: Exemplo de características biométricas.

identificação, o sistema biométrico visa identificar o indivíduo a quem correspondem as características biométricas coletadas pelo sensor, dentre todos aqueles presentes no banco de dados. A Figura 10 mostra os principais elementos de um sistema biométrico típico.

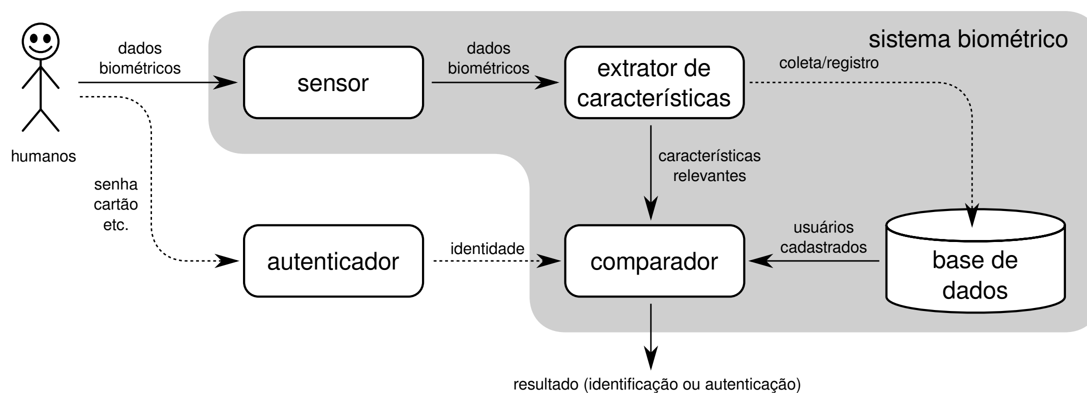

Figura 10: Um sistema biométrico típico.

4.6 Desafio-resposta

Em algumas situações o uso de senhas é indesejável, pois sua exposição indevida pode comprometer a segurança do sistema. Um exemplo disso são os serviços via rede: caso o tráfego de rede possa ser capturado por um intruso, este terá acesso às senhas transmitidas entre o cliente e o servidor. Uma técnica interessante para resolver esse problema são os protocolos de *desafio-resposta*.

A técnica de desafio-resposta se baseia sobre um segredo s previamente definido entre o cliente e o servidor (ou o usuário e o sistema), que pode ser uma senha ou uma chave criptográfica, e um algoritmo de cifragem ou resumo $hash(x)$, também previamente definido. No início da autenticação, o servidor escolhe um valor aleatório d e o envia ao cliente, como um *desafio*. O cliente recebe esse desafio, o concatena com seu segredo s, calcula o resumo da concatenação e a devolve ao servidor, como *resposta* ($r = hash(s + d)$). O servidor executa a mesma operação de seu lado, usando o valor do segredo armazenado localmente (s') e compara o resultado obtido $r' = hash(s' + d)$ com

a resposta r fornecida pelo cliente. Se ambos os resultados forem iguais, os segredos são iguais ($r = r' \Rightarrow s = s'$) e o cliente é considerado autêntico. A Figura 11 apresenta os passos desse algoritmo.

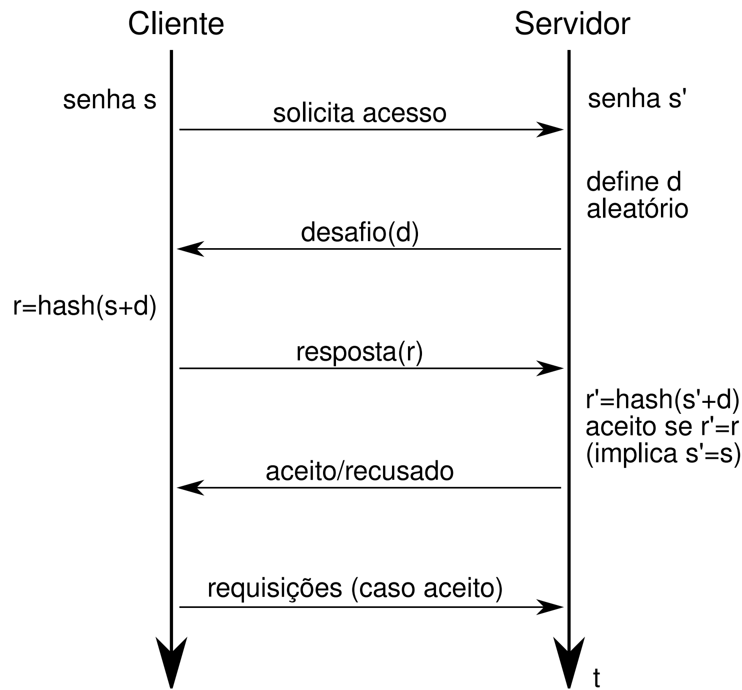

Figura 11: Autenticação por desafio-resposta.

A estratégia de desafio-resposta é robusta, porque o segredo s nunca é exposto fora do cliente nem do servidor; além disso, como o desafio d é aleatório e a resposta é cifrada, intrusos que eventualmente conseguirem capturar d ou r não poderão utilizá-los para se autenticar nem para descobrir s. Variantes dessa técnica são usadas em vários protocolos de rede.

4.7 Certificados de autenticação

Uma forma cada vez mais frequente de autenticação envolve o uso de *certificados digitais*. Conforme apresentado na Seção 3.5, um certificado digital é um documento assinado digitalmente, através de técnicas de criptografia assimétrica e resumo criptográfico. Os padrões de certificados PGP e X.509 definem certificados de autenticação (ou de identidade), cujo objetivo é identificar entidades através de suas chaves públicas. Um certificado de autenticação conforme o padrão X.509 contém as seguintes informações [Mollin, 2000]:

- Número de versão do padrão X.509 usado no certificado;
- Chave pública do proprietário do certificado e indicação do algoritmo de criptografia ao qual ela está associada e eventuais parâmetros;
- Número serial único, definido pelo emissor do certificado (quem o assinou);

- Identificação detalhada do proprietário do certificado, definida de acordo com normas do padrão X.509;
- Período de validade do certificado (datas de início e final de validade);
- Identificação da Autoridade Certificadora que emitiu/assinou o certificado;
- Assinatura digital do certificado e indicação do algoritmo usado na assinatura e eventuais parâmetros;

Os certificados digitais são o principal mecanismo usado para verificar a autenticidade de serviços acessíveis através da Internet, como bancos e comércio eletrônico. Nesse caso, eles são usados para autenticar os sistemas para os usuários. No entanto, é cada vez mais frequente o uso de certificados para autenticar os próprios usuários. Nesse caso, um *smartcard* ou um dispositivo USB contendo o certificado é conectado ao sistema para permitir a autenticação do usuário.

4.8 Kerberos

O sistema de autenticação *Kerberos* foi proposto pelo MIT nos anos 80 [Neuman and Ts'o, 1994]. Hoje, esse sistema é utilizado para centralizar a autenticação de rede em vários sistemas operacionais, como Windows, Solaris, MacOS X e Linux. O sistema Kerberos se baseia na noção de *tickets*, que são obtidos pelos clientes junto a um serviço de autenticação e podem ser usados para acessar os demais serviços da rede. Os tickets são cifrados usando criptografia simétrica DES e têm validade limitada, para aumentar sua segurança.

Os principais componentes de um sistema Kerberos são o Serviço de Autenticação (AS - *Authentication Service*), o Serviço de Concessão de Tickets (TGS - *Ticket Granting Service*), a base de chaves, os clientes e os serviços de rede que os clientes podem acessar. Juntos, o AS e o TGS constituem o *Centro de Distribuição de Chaves* (KDC - *Key Distribution Center*). O funcionamento básico do sistema Kerberos, ilustrado na Figura 12, é relativamente simples: o cliente se autentica junto ao AS (passo 1) e obtém um ticket de acesso ao serviço de tickets TGS (passo 2). A seguir, solicita ao TGS um ticket de acesso ao servidor desejado (passos 3 e 4). Com esse novo ticket, ele pode se autenticar junto ao servidor desejado e solicitar serviços (passos 5 e 6).

No Kerberos, cada cliente c possui uma chave secreta k_c registrada no servidor de autenticação AS. Da mesma forma, cada servidor s também tem sua chave k_s registrada no AS. As chaves são simétricas, usando cifragem DES, e somente são conhecidas por seus respectivos proprietários e pelo AS. Os seguintes passos detalham o funcionamento do Kerberos versão 5 [Neuman and Ts'o, 1994]:

1. Uma máquina cliente c desejando acessar um determinado servidor s envia uma solicitação de autenticação ao serviço de autenticação (AS); essa mensagem m_1 contém sua identidade (c), a identidade do serviço desejado (tgs), um prazo de validade solicitado (ts) e um número aleatório (n_1) que será usado para verificar se a resposta do AS corresponde ao pedido efetuado:

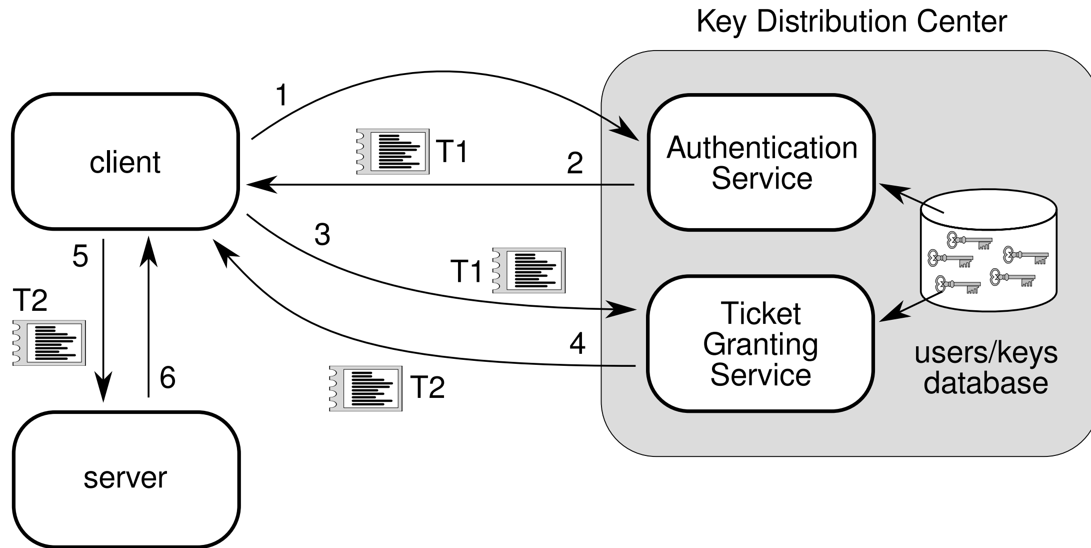

Figura 12: Visão geral do serviço Kerberos.

$$m_1 = [c \ \ tgs \ \ ts \ \ n_1]$$

2. A resposta do AS (mensagem m_2) contém duas partes: a primeira parte contém a chave de sessão a ser usada na comunicação com o TGS (k_{c-tgs}) e o número aleatório n_1, ambos cifrados com a chave do cliente k_c registrada no AS; a segunda parte é um ticket cifrado com a chave do TGS (k_{tgs}), contendo a identidade do cliente (c), o prazo de validade do ticket concedido pelo AS (tv) e uma chave de sessão k_{c-tgs}, a ser usada na interação com o TGS:

$$m_2 = [\{k_{c-tgs} \ \ n_1\}_{k_c} \ \ T_{c-tgs}] \quad \text{onde } T_{c-tgs} = \{c \ \ tv \ \ k_{c-tgs}\}_{k_{tgs}}$$

   O ticket T_{c-tgs} fornecido pelo AS para permitir o acesso ao TGS é chamado TGT (*Ticket Granting Ticket*), e possui um prazo de validade limitado (geralmente de algumas horas). Ao receber m_2, o cliente tem acesso à chave de sessão k_{c-tgs} e ao ticket TGT. Todavia, esse ticket é cifrado com a chave k_{tgs} e portanto somente o TGS poderá abrí-lo.

3. A seguir, o cliente envia uma solicitação ao TGS (mensagem m_3) para obter um ticket de acesso ao servidor desejado s. Essa solicitação contém a identidade do cliente (c) e a data atual (t), ambos cifrados com a chave de sessão k_{c-tgs}, o ticket TGT recebido em m_2, a identidade do servidor s e um número aleatório n_2:

$$m_3 = [\{c \ \ t\}_{k_{c-tgs}} \ \ T_{c-tgs} \ \ s \ \ n_2]$$

4. Após verificar a validade do ticket TGT, o TGS devolve ao cliente uma mensagem m_4 contendo a chave de sessão k_{c-s} a ser usada no acesso ao servidor s e o número

aleatório n_2 informado em m_3, ambos cifrados com a chave de sessão k_{c-tgs}, e um ticket T_{c-s} cifrado, que deve ser apresentado ao servidor s:

$$m_4 = [\{k_{c-s} \;\; n\}_{k_{c-tgs}} \;\; T_{c-s}] \quad \text{onde } T_{c-s} = \{c \;\; tv \;\; k_{c-s}\}_{k_s}$$

5. O cliente usa a chave de sessão k_{c-s} e o ticket T_{c-s} para se autenticar junto ao servidor s através da mensagem m_5. Essa mensagem contém a identidade do cliente (c) e a data atual (t), ambos cifrados com a chave de sessão k_{c-s}, o ticket T_{c-s} recebido em m_4 e o pedido de serviço ao servidor (*request*), que é dependente da aplicação:

$$m_5 = [\{c \;\; t\}_{k_{c-s}} \;\; T_{c-s} \;\; request]$$

6. Ao receber m_5, o servidor s decifra o ticket T_{c-s} para obter a chave de sessão k_{c-s} e a usa para decifrar a primeira parte da mensagem e confirmar a identidade do cliente. Feito isso, o servidor pode atender a solicitação e responder ao cliente, cifrando sua resposta com a chave de sessão k_{c-s}:

$$m_6 = [\{reply\}_{k_{c-s}}]$$

Enquanto o ticket de serviço T_{c-s} for válido, o cliente pode enviar solicitações ao servidor sem a necessidade de se reautenticar. Da mesma forma, enquanto o ticket T_{c-tgs} for válido, o cliente pode solicitar tickets de acesso a outros servidores sem precisar se reautenticar. Pode-se observar que em nenhum momento as chaves de sessão k_{c-tgs} e k_{c-s} circularam em aberto através da rede. Além disso, a presença de prazos de validade para as chaves permite minimizar os riscos de uma eventual captura da chave. Informações mais detalhadas sobre o funcionamento do protocolo Kerberos 5 podem ser encontradas em [Neuman et al., 2005].

4.9 Infra-estruturas de autenticação

A autenticação é um procedimento necessário em vários serviços de um sistema computacional, que vão de simples sessões de terminal em modo texto a serviços de rede, como e-mail, bancos de dados e terminais gráficos remotos. Historicamente, cada forma de acesso ao sistema possuía seus próprios mecanismos de autenticação, com suas próprias regras e informações. Essa situação dificultava a criação de novos serviços, pois estes deveriam também definir seus próprios métodos de autenticação. Além disso, a existência de vários mecanismos de autenticação desconexos prejudicava a experiência do usuário e dificultava a gerência do sistema.

Para resolver esse problema, foram propostas infra-estruturas de autenticação (*authentication frameworks*) que unificam as técnicas de autenticação, oferecem uma interface de programação homogênea e usam as mesmas informações (pares *login/senha*, dados biométricos, certificados, etc.). Assim, as informações de autenticação são coerentes

entre os diversos serviços, novas técnicas de autenticação podem ser automaticamente usadas por todos os serviços e, sobretudo, a criação de novos serviços é simplificada.

A visão genérica de uma infra-estrutura de autenticação é apresentada na Figura 13. Nela, os vários mecanismos disponíveis de autenticação são oferecidos às aplicações através de uma interface de programação (API) padronizada. As principais infra-estruturas de autenticação em uso nos sistemas operacionais atuais são:

PAM (*Pluggable Authentication Modules*): proposto inicialmente para o sistema Solaris, foi depois adotado em vários outros sistema UNIX, como FreeBSD, NetBSD, MacOS X e Linux;

XSSO (*X/Open Single Sign-On*): é uma tentativa de extensão e padronização do sistema PAM, ainda pouco utilizada;

BSD Auth : usada no sistema operacional OpenBSD; cada método de autenticação é implementado como um processo separado, respeitando o princípio do privilégio mínimo (vide Seção 5.1);

NSS (*Name Services Switch*): infra-estrutura usada em sistemas UNIX para definir as bases de dados a usar para vários serviços do sistema operacional, inclusive a autenticação;

GSSAPI (*Generic Security Services* API): padrão de API para acesso a serviços de segurança, como autenticação, confidencialidade e integridade de dados;

SSPI (*Security Support Provider Interface*): variante proprietária da GSSAPI, específica para plataformas Windows.

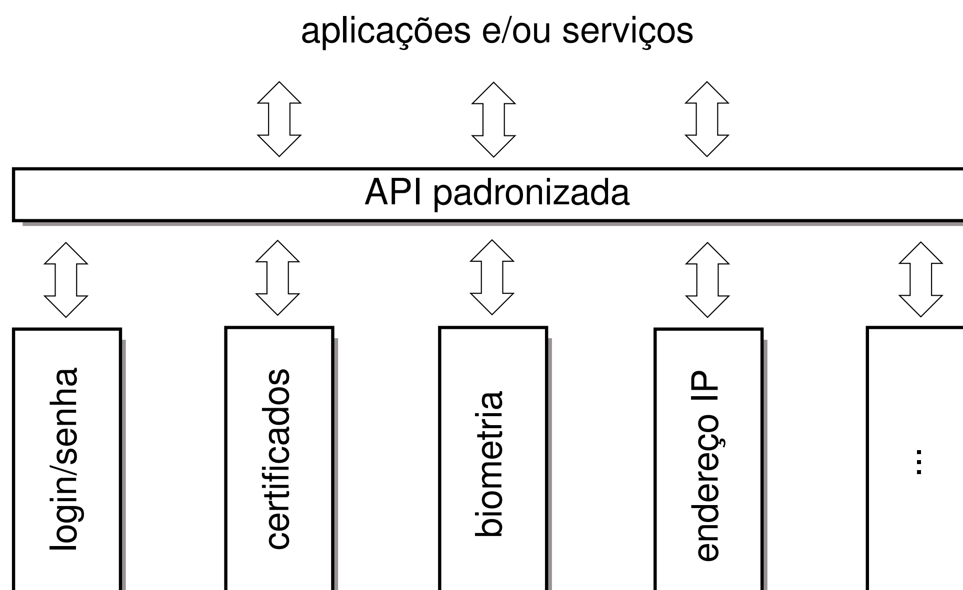

Figura 13: Estrutura genérica de uma infra-estrutura de autenticação.

5 Controle de acesso

Em um sistema computacional, o controle de acesso consiste em mediar cada solicitação de acesso de um usuário autenticado a um recurso ou dado mantido pelo sistema, para determinar se aquela solicitação deve ser autorizada ou negada [Samarati and De Capitani di Vimercati, 2001]. Praticamente todos os recursos de um sistema operacional típico estão submetidos a um controle de acesso, como arquivos, áreas de memória, semáforos, portas de rede, dispositivos de entrada/saída, etc. Há alguns conceitos importantes para a compreensão do controle de acesso, como políticas, modelos e mecanismos. Esses conceitos serão estudados nesta seção.

Em controle de acesso, é habitual classificar as entidades de um sistema em dois grupos: os *sujeitos* e os *objetos*. Sujeitos são todas aquelas entidades que exercem um papel ativo no sistema, como processos, *threads* ou transações. Normalmente um sujeito opera em nome de um usuário, que pode ser um ser humano ou outro sistema computacional externo. Objetos são as entidades passivas utilizadas pelos sujeitos, como arquivos, áreas de memória ou registros em um banco de dados. Em alguns casos, um sujeito pode ser visto como objeto por outro sujeito (por exemplo, quando um sujeito deve enviar uma mensagem a outro sujeito). Tanto sujeitos quanto objetos podem ser organizados em grupos e hierarquias, para facilitar a gerência da segurança.

5.1 Políticas, modelos e mecanismos de controle de acesso

Uma *política de controle de acesso* é uma visão abstrata das possibilidades de acesso a recursos (objetos) pelos usuários (sujeitos) de um sistema. Essa política consiste basicamente de um conjunto de regras definindo os acessos possíveis aos recursos do sistema e eventuais condições necessárias para permitir cada acesso. Por exemplo, as regras a seguir poderiam constituir parte da política de segurança de um sistema de informações médicas:

- Médicos podem consultar os prontuários de seus pacientes;
- Médicos podem modificar os prontuários de seus pacientes enquanto estes estiverem internados;
- O supervisor geral pode consultar os prontuários de todos os pacientes;
- Enfermeiros podem consultar apenas os prontuários dos pacientes de sua seção e somente durante seu período de turno;
- Assistentes não podem consultar prontuários;
- Prontuários de pacientes de planos de saúde privados podem ser consultados pelo responsável pelo respectivo plano de saúde no hospital;
- Pacientes podem consultar seus próprios prontuários (aceitar no máximo 30 pacientes simultâneos).

As regras ou definições individuais de uma política são denominadas *autorizações*. Uma política de controle de acesso pode ter autorizações baseadas em *identidades* (como sujeitos e objetos) ou em outros *atributos* (como idade, sexo, tipo, preço, etc.); as autorizações podem ser *individuais* (a sujeitos) ou *coletivas* (a grupos); também podem existir autorizações *positivas* (permitindo o acesso) ou *negativas* (negando o acesso); por fim, uma política pode ter autorizações dependentes de *condições externas* (como o tempo ou a carga do sistema). Além da política de acesso aos objetos, também deve ser definida uma *política administrativa*, que define quem pode modificar/gerenciar as políticas vigentes no sistema [Samarati and De Capitani di Vimercati, 2001].

O conjunto de autorizações de uma política deve ser ao mesmo tempo *completo*, cobrindo todas as possibilidades de acesso que vierem a ocorrer no sistema, e *consistente*, sem regras conflitantes entre si (por exemplo, uma regra que permita um acesso e outra que negue esse mesmo acesso). Além disso, toda política deve buscar respeitar o *princípio do privilégio mínimo* [Saltzer and Schroeder, 1975], segundo o qual um usuário nunca deve receber mais autorizações que aquelas que necessita para cumprir sua tarefa. A construção e validação de políticas de controle de acesso é um tema complexo, que está fora do escopo deste texto, sendo melhor descrito em [di Vimercati et al., 2005, di Vimercati et al., 2007].

As políticas de controle de acesso definem de forma abstrata como os sujeitos podem acessar os objetos do sistema. Existem muitas formas de se definir uma política, que podem ser classificadas em quatro grandes classes: políticas *discricionárias*, políticas *obrigatórias*, políticas *baseadas em domínios* e políticas *baseadas em papéis* [Samarati and De Capitani di Vimercati, 2001]. As próximas seções apresentam com mais detalhe cada uma dessas classes de políticas.

Geralmente a descrição de uma política de controle de acesso é muito abstrata e informal. Para sua implementação em um sistema real, ela precisa ser descrita de uma forma precisa, através de um *modelo de controle de acesso*. Um modelo de controle de acesso é uma representação lógica ou matemática da política, de forma a facilitar sua implementação e permitir a análise de eventuais erros. Em um modelo de controle de acesso, as autorizações de uma política são definidas como relações lógicas entre *atributos do sujeito* (como seus identificadores de usuário e grupo) *atributos do objeto* (como seu caminho de acesso ou seu proprietário) e eventuais condições externas (como o horário ou a carga do sistema). Nas próximas seções, para cada classe de políticas de controle de acesso apresentada serão discutidos alguns modelos aplicáveis à mesma.

Por fim, os *mecanismos de controle de acesso* são as estruturas necessárias à implementação de um determinado modelo em um sistema real. Como é bem sabido, é de fundamental importância a separação entre políticas e mecanismos, para permitir a substituição ou modificação de políticas de controle de acesso de um sistema sem incorrer em custos de modificação de sua implementação. Assim, um mecanismo de controle de acesso ideal deveria ser capaz de suportar qualquer política de controle de acesso.

5.2 Políticas discricionárias

As políticas discricionárias (DAC - *Discretionary Access Control*) se baseiam na atribuição de permissões de forma individualizada, ou seja, pode-se claramente conceder (ou negar) a um sujeito específico s a permissão de executar a ação a sobre um objeto específico o. Em sua forma mais simples, as regras de uma política discricionária têm a forma $\langle s, o, +a\rangle$ ou $\langle s, o, -a\rangle$, para respectivamente autorizar ou negar a ação a do sujeito s sobre o objeto o (também podem ser definidas regras para grupos de usuários e/ou de objetos devidamente identificados). Por exemplo:

- O usuário `Beto` pode ler e escrever arquivos em `/home/beto`
- Usuários do grupo `admin` podem ler os arquivos em `/suporte`

O responsável pela administração das permissões de acesso a um objeto pode ser o seu proprietário ou um administrador central. A definição de quem estabelece as regras da política de controle de acesso é inerente a uma política administrativa, independente da política de controle de acesso em si[6].

5.2.1 Matriz de controle de acesso

O modelo matemático mais simples e conveniente para representar políticas discricionárias é a *Matriz de Controle de Acesso*, proposta em [Lampson, 1971]. Nesse modelo, as autorizações são dispostas em uma matriz, cujas linhas correspondem aos sujeitos do sistema e cujas colunas correspondem aos objetos. Em termos formais, considerando um conjunto de sujeitos $\mathbb{S} = \{s_1, s_2, \ldots, s_m\}$, um conjunto de objetos $\mathbb{O} = \{o_1, o_2, \ldots, o_n\}$ e um conjunto de ações possíveis sobre os objetos $\mathbb{A} = \{a_1, a_2, \ldots, a_p\}$, cada elemento M_{ij} da matriz de controle de acesso é um sub-conjunto (que pode ser vazio) do conjunto de ações possíveis, que define as ações que $s_i \in \mathbb{S}$ pode efetuar sobre $o_j \in \mathbb{O}$:

$$\forall s_i \in \mathbb{S}, \forall o_j \in \mathbb{O}, M_{ij} \subseteq \mathbb{A}$$

Por exemplo, considerando um conjunto de sujeitos $\mathbb{S} = \{Alice, Beto, Carol, Davi\}$, um conjunto de objetos $\mathbb{O} = \{file_1, file_2, program_1, socket_1\}$ e um conjunto de ações $\mathbb{A} = \{read, write, execute, remove\}$, podemos ter uma matriz de controle de acesso como a apresentada na Tabela 1.

Apesar de simples, o modelo de matriz de controle de acesso é suficientemente flexível para suportar políticas administrativas. Por exemplo, considerando uma política administrativa baseada na noção de proprietário do recurso, poder-se-ia considerar que que cada objeto possui um ou mais proprietários (*owner*), e que os sujeitos podem modificar as entradas da matriz de acesso relativas aos objetos que possuem. Uma matriz de controle de acesso com essa política administrativa é apresentada na Tabela 2.

[6] Muitas políticas de controle de acesso discricionárias são baseadas na noção de que cada recurso do sistema possui um proprietário, que decide quem pode acessar o recurso. Isso ocorre por exemplo nos sistemas de arquivos, onde as permissões de acesso a cada arquivo ou diretório são definidas pelo respectivo proprietário. Contudo, a noção de "proprietário" de um recurso não é essencial para a construção de políticas discricionárias [Shirey, 2000].

	$file_1$	$file_2$	$program_1$	$socket_1$
Alice	*read* *write* *remove*	*read* *write*	*execute*	*write*
Beto	*read* *write*	*read* *write* *remove*	*read*	
Carol		*read*	*execute*	*read* *write*
Davi	*read*	*append*	*read*	*read* *append*

Tabela 1: Uma matriz de controle de acesso

	$file_1$	$file_2$	$program_1$	$socket_1$
Alice	*read* *write* *remove* ***owner***	*read* *write*	*execute*	*write*
Beto	*read* *write*	*read* *write* *remove* ***owner***	*read* ***owner***	
Carol		*read*	*execute*	*read* *write*
Davi	*read*	*write*	*read*	*read* *write* ***owner***

Tabela 2: Uma matriz de controle de acesso com política administrativa

Embora seja um bom modelo conceitual, a matriz de acesso é inadequada para implementação. Em um sistema real, com milhares de sujeitos e milhões de objetos, essa matriz pode se tornar gigantesca e consumir muito espaço. Como em um sistema real cada sujeito tem seu acesso limitado a um pequeno grupo de objetos (e vice-versa), a matriz de acesso geralmente é esparsa, ou seja, contém muitas células vazias. Assim, algumas técnicas simples podem ser usadas para implementar esse modelo, como as tabelas de autorizações, as listas de controle de acesso e as listas de capacidades [Samarati and De Capitani di Vimercati, 2001], explicadas a seguir.

5.2.2 Tabela de autorizações

Na abordagem conhecida como **Tabela de Autorizações**, as entradas não-vazias da matriz são relacionadas em uma tabela com três colunas: *sujeitos*, *objetos* e *ações*, onde cada tupla da tabela corresponde a uma autorização. Esta abordagem é muito utilizada em sistemas gerenciadores de bancos de dados (DBMS - *Database Management Systems*), devido à sua facilidade de implementação e consulta nesse tipo de ambiente. A Tabela 3 mostra como ficaria a matriz de controle de acesso da Tabela 2 sob a forma de uma tabela de autorizações.

5.2.3 Listas de controle de acesso

Outra abordagem usual é a **Lista de Controle de Acesso**. Nesta abordagem, para cada objeto é definida uma lista de controle de acesso (ACL - *Access Control List*), que contém a relação de sujeitos que podem acessá-lo, com suas respectivas permissões. Cada lista de controle de acesso corresponde a uma coluna da matriz de controle de acesso. Como exemplo, as listas de controle de acesso relativas à matriz de controle de acesso da Tabela 2 seriam:

$$
\begin{aligned}
ACL(file_1) &= \{ \quad Alice : (read, write, remove, owner),\\
&\qquad Beto : (read, write),\\
&\qquad Davi : (read) \}\\
ACL(file_2) &= \{ \quad Alice : (read, write),\\
&\qquad Beto : (read, write, remove, owner),\\
&\qquad Carol : (read),\\
&\qquad Davi : (write) \}\\
ACL(program_1) &= \{ \quad Alice : (execute),\\
&\qquad Beto : (read, owner),\\
&\qquad Carol : (execute),\\
&\qquad Davi : (read) \}\\
ACL(socket_1) &= \{ \quad Alice : (write),\\
&\qquad Carol : (read, write),\\
&\qquad Davi : (read, write, owner) \}
\end{aligned}
$$

Sujeito	**Objeto**	**Ação**
Alice	$file_1$	*read*
Alice	$file_1$	*write*
Alice	$file_1$	*remove*
Alice	$file_1$	*owner*
Alice	$file_2$	*read*
Alice	$file_2$	*write*
Alice	$program_1$	*execute*
Alice	$socket_1$	*write*
Beto	$file_1$	*read*
Beto	$file_1$	*write*
Beto	$file_2$	*read*
Beto	$file_2$	*write*
Beto	$file_2$	*remove*
Beto	$file_2$	*owner*
Beto	$program_1$	*read*
Beto	$socket_1$	*owner*
Carol	$file_2$	*read*
Carol	$program_1$	*execute*
Carol	$socket_1$	*read*
Carol	$socket_1$	*write*
Davi	$file_1$	*read*
Davi	$file_2$	*write*
Davi	$program_1$	*read*
Davi	$socket_1$	*read*
Davi	$socket_1$	*write*
Davi	$socket_1$	*owner*

Tabela 3: Tabela de autorizações

Esta forma de implementação é a mais frequentemente usada em sistemas operacionais, por ser simples de implementar e bastante robusta. Por exemplo, o sistema de arquivos associa uma ACL a cada arquivo ou diretório, para indicar quem são os sujeitos autorizados a acessá-lo. Em geral, somente o proprietário do arquivo pode modificar sua ACL, para incluir ou remover permissões de acesso.

5.2.4 Listas de capacidades

Uma terceira abordagem possível para a implementação da matriz de controle de acesso é a **Lista de Capacidades** (CL - *Capability List*), ou seja, uma lista de objetos que um dado sujeito pode acessar e suas respectivas permissões sobre os mesmos. Cada lista de capacidades corresponde a uma linha da matriz de acesso. Como exemplo, as listas de capacidades correspondentes à matriz de controle de acesso da Tabela 2 seriam:

$$
\begin{aligned}
CL(Alice) = \{ \; & file_1 : (read, write, remove, owner), \\
& file_2 : (read, write), \\
& program_1 : (execute), \\
& socket_1 : (write) \; \} \\
CL(Beto) = \{ \; & file_1 : (read, write), \\
& file_2 : (read, write, remove, owner), \\
& program_1 : (read, owner) \; \} \\
CL(Carol) = \{ \; & file_2 : (read), \\
& program_1 : (execute), \\
& socket_1 : (read, write) \; \} \\
CL(Davi) = \{ \; & file_1 : (read), \\
& file_2 : (write), \\
& program_1 : (read), \\
& socket_1 : (read, write, owner) \; \}
\end{aligned}
$$

Uma capacidade pode ser vista como uma ficha ou *token*: sua posse dá ao proprietário o direito de acesso ao objeto em questão. Capacidades são pouco usadas em sistemas operacionais, devido à sua dificuldade de implementação e possibilidade de fraude, pois uma capacidade mal implementada pode ser transferida deliberadamente a outros sujeitos, ou modificada pelo próprio proprietário para adicionar mais permissões a ela. Outra dificuldade inerente às listas de capacidades é a administração das autorizações: por exemplo, quem deve ter permissão para modificar uma lista de capacidades, e como retirar uma permissão concedida anteriormente a um sujeito? Alguns sistemas operacionais que implementam o modelo de capacidades são discutidos na Seção 5.6.4.

5.3 Políticas obrigatórias

Nas *políticas obrigatórias* (MAC - *Mandatory Access Control*) o controle de acesso é definido por regras globais incontornáveis, que não dependem das identidades dos sujeitos e objetos nem da vontade de seus proprietários ou mesmo do administrador do sistema [Samarati and De Capitani di Vimercati, 2001]. Essas regras são normalmente baseadas em atributos dos sujeitos e/ou dos objetos, como mostram estes exemplos bancários (fictícios):

- Cheques com valor acima de R$ 5.000,00 devem ser necessariamente depositados e não podem ser descontados;
- Clientes com renda mensal acima de R$3.000,00 não têm acesso ao crédito consignado.

Uma das formas mais usuais de política obrigatória são as *políticas multi-nível* (MLS - *Multi-Level Security*), que se baseiam na classificação de sujeitos e objetos do sistema em *níveis de segurança* (*clearance levels*) e na definição de regras usando esses níveis. Um exemplo bem conhecido de escala de níveis de classificação é aquela usada pelo governo britânico para definir a confidencialidade de um documento:

- TS: *Top Secret* (*Ultrassecreto*)
- S: *Secret* (*Secreto*)
- C: *Confidential* (*Confidencial*)
- R: *Restrict* (*Reservado*)
- U: *Unclassified* (*Público*)

Em uma política MLS, considera-se que os níveis de segurança estão ordenados entre si (por exemplo, $U < R < C < S < TS$) e são associados a todos os sujeitos e objetos do sistema, sob a forma de *habilitação* dos sujeitos ($h(s_i)$) e *classificação* dos objetos ($c(o_j)$). As regras da política são então estabelecidas usando essas habilitações e classificações, como mostram os modelos descritos a seguir.

5.3.1 Modelo de Bell-LaPadula

Um modelo de controle de acesso que permite formalizar políticas multi-nível é o de *Bell-LaPadula* [Bell and LaPadula, 1974], usado para garantir a confidencialidade das informações. Esse modelo consiste basicamente de duas regras:

No-Read-Up ("não ler acima", ou "propriedade simples"): impede que um sujeito leia objetos que se encontrem em níveis de segurança acima do seu. Por exemplo, um sujeito habilitado como confidencial (C) somente pode ler objetos cuja classificação seja confidencial (C), reservada (R) ou pública (U). Considerando um sujeito s e um objeto o, formalmente temos:

$$request(s, o, read) \iff h(s) \geq c(o)$$

No-Write-Down (“não escrever abaixo”, ou “propriedade ⋆”): impede que um sujeito escreva em objetos abaixo de seu nível de segurança, para evitar o “vazamento” de informações dos níveis superiores para os inferiores. Por exemplo, um sujeito habilitado como confidencial somente pode escrever em objetos cuja classificação seja confidencial, secreta ou ultrassecreta. Formalmente, temos:

$$request(s, o, write) \iff h(s) \leq c(o)$$

Pode-se perceber facilmente que a política obrigatória representada pelo modelo de Bell-LaPadula visa proteger a *confidencialidade* das informações do sistema, evitando que estas fluam dos níveis superiores para os inferiores. Todavia, nada impede um sujeito com baixa habilitação escrever sobre um objeto de alta classificação, destruindo seu conteúdo.

5.3.2 Modelo de Biba

Para garantir a *integridade* das informações, um modelo dual ao de Bell-LaPadula foi proposto por Biba [Biba, 1977]. Esse modelo define níveis de integridade $i(x)$ para sujeitos e objetos (como *Baixa*, *Média*, *Alta* e *Sistema*, com $B < M < A < S$), e também possui duas regras básicas:

No-Write-Up (“não escrever acima”, ou “propriedade simples de integridade”): impede que um sujeito escreva em objetos acima de seu nível de integridade, preservando-os íntegros. Por exemplo, um sujeito de integridade média (M) somente pode escrever em objetos de integridade baixa (B) ou média (M). Formalmente, temos:

$$request(s, o, write) \iff i(s) \geq i(o)$$

No-Read-Down (“não ler abaixo”, ou “propriedade ⋆ de integridade”): impede que um sujeito leia objetos em níveis de integridade abaixo do seu, para não correr o risco de ler informação duvidosa. Por exemplo, um sujeito com integridade alta (A) somente pode ler objetos com integridade alta (A) ou de sistema (S). Formalmente, temos:

$$request(s, o, read) \iff i(s) \leq i(o)$$

A política obrigatória definida através do modelo de Biba evita violações de integridade, mas não garante a confidencialidade das informações. Para que as duas políticas (confidencialidade e integridade) possam funcionar em conjunto, é necessário portanto associar a cada sujeito e objeto do sistema um nível de confidencialidade e um nível de integridade, possivelmente distintos.

É importante observar que, na maioria dos sistemas reais, **as políticas obrigatórias não substituem as políticas discricionárias**, mas as complementam. Em um sistema que usa políticas obrigatórias, cada acesso a recurso é verificado usando a politica obrigatória e também uma politica discricionária; o acesso é permitido somente se

ambas as politicas o autorizarem. A ordem de avaliação das políticas MAC e DAC obviamente não afeta o resultado final, mas pode ter impacto sobre o desempenho do sistema. Por isso, deve-se primeiro avaliar a política mais restritiva, ou seja, aquela que tem mais probabilidades de negar o acesso.

5.3.3 Categorias

Uma extensão frequente às políticas multi-nível é a noção de *categorias* ou *compartimentos*. Uma categoria define uma área funcional dentro do sistema computacional, como "pessoal", "projetos", "financeiro", "suporte", etc. Normalmente o conjunto de categorias é estático não há uma ordem hierárquica entre elas. Cada sujeito e cada objeto do sistema são "rotulados" com uma ou mais categorias; a política então consiste em restringir o acesso de um sujeito somente aos objetos pertencentes às mesmas categorias dele, ou a um sub-conjunto destas. Dessa forma, um sujeito com as categorias $\{suporte, financeiro\}$ só pode acessar objetos rotulados como $\{suporte, financeiro\}$, $\{suporte\}$, $\{financeiro\}$ ou $\{\phi\}$. Formalmente: sendo $\mathbb{C}(s)$ o conjunto de categorias associadas a um sujeito s e $\mathbb{C}(o)$ o conjunto de categorias associadas a um objeto o, s só pode acessar o se $\mathbb{C}(s) \supseteq \mathbb{C}(o)$ [Samarati and De Capitani di Vimercati, 2001].

5.4 Políticas baseadas em domínios e tipos

O *domínio de segurança* de um sujeito define o conjunto de objetos que ele pode acessar e como pode acessá-los. Muitas vezes esse domínio está definido implicitamente nas regras das políticas obrigatórias ou na matriz de controle de acesso de uma política discricionária. As *políticas baseadas em domínios e tipos* (DTE - *Domain/Type Enforcement policies*) [Boebert and Kain, 1985] tornam explícito esse conceito: cada sujeito s do sistema é rotulado com um atributo constante definindo seu domínio $domain(s)$ e cada objeto o é associado a um tipo $type(o)$, também constante.

No modelo de implementação de uma política DTE definido em [Badger et al., 1995], as permissões de acesso de sujeitos a objetos são definidas em uma tabela global chamada *Tabela de Definição de Domínios* (DDT - *Domain Definition Table*), na qual cada linha é associada a um domínio e cada coluna a um tipo; cada célula $DDT[x, y]$ contém as permissões de sujeitos do domínio x a objetos do tipo y:

$$request(s, o, action) \iff action \in DDT[domain(s), type(o)]$$

Por sua vez, as interações entre sujeitos (trocas de mensagens, sinais, etc.) são reguladas através de uma *Tabela de Interação entre Domínios* (DIT - *Domain Interaction Table*). Nessa tabela, linhas e colunas correspondem a domínios e cada célula $DIT[x, y]$ contém as interações possíveis de um sujeito no domínio x sobre um sujeito no domínio y:

$$request(s_i, s_j, interaction) \iff interaction \in DIT[domain(s_i), domain(s_j)]$$

Eventuais mudanças de domínio podem ser associadas a programas executáveis rotulados como *pontos de entrada* (*entry points*). Quando um processo precisa mudar de

domínio, ele executa o ponto de entrada correspondente ao domínio de destino, se tiver permissão para tal.

O código a seguir define uma política de controle de acesso DTE, usada como exemplo em [Badger et al., 1995]. Essa política está representada graficamente (de forma simplificada) na Figura 14.

```
/* type definitions */
type unix_t,     /* normal UNIX files, programs, etc. */
     specs_t,    /* engineering specifications */
     budget_t,   /* budget projections */
     rates_t;    /* labor rates */

#define DEFAULT (/bin/sh), (/bin/csh), (rxd->unix_t) /* macro */

/* domain definitions */
domain engineer_d = DEFAULT, (rwd->specs_t);
domain project_d = DEFAULT, (rwd->budget_t), (rd->rates_t);
domain accounting_d = DEFAULT, (rd->budget_t), (rwd->rates_t);
domain system_d  = (/etc/init), (rwxd->unix_t), (auto->login_d);
domain login_d   = (/bin/login), (rwxd->unix_t),
                   (exec-> engineer_d, project_d, accounting_d);

initial_domain system_d; /* system starts in this domain */

/* assign resources (files and directories) to types */
assign -r unix_t   /;        /* default for all files */
assign -r specs_t  /projects/specs;
assign -r budget_t /projects/budget;
assign -r rates_t  /projects/rates;
```

A implementação direta desse modelo sobre um sistema real pode ser inviável, pois exige a classificação de todos os sujeitos e objetos do mesmo em domínios e tipos. Para atenuar esse problema, [Badger et al., 1995, Cowan et al., 2000] propõem o uso de *tipagem implícita*: todos os objetos que satisfazem um certo critério (como por exemplo ter como caminho `/usr/local/*`) são automaticamente classificados em um dado tipo. Da mesma forma, os domínios podem ser definidos pelos nomes dos programas executáveis que os sujeitos executam (como `/usr/bin/httpd` e `/usr/lib/httpd/plugin/*` para o domínio do servidor Web). Além disso, ambos os autores propõem linguagens para a definição dos domínios e tipos e para a descrição das políticas de controle de acesso.

5.5 Políticas baseadas em papéis

Um dos principais problemas de segurança em um sistema computacional é a administração correta das políticas de controle de acesso. As políticas MAC são geralmente consideradas pouco flexíveis e por isso as políticas DAC acabam sendo muito mais usadas. Todavia, gerenciar as autorizações à medida em que usuários mudam de cargo e assumem novas responsabilidades, novos usuários entram na empresa e outros saem pode ser uma tarefa muito complexa e sujeita a erros.

Esse problema pode ser reduzido através do *controle de acesso baseado em papéis* (RBAC - *Role-Based Access Control*) [Sandhu et al., 1996]. Uma política RBAC define um conjunto

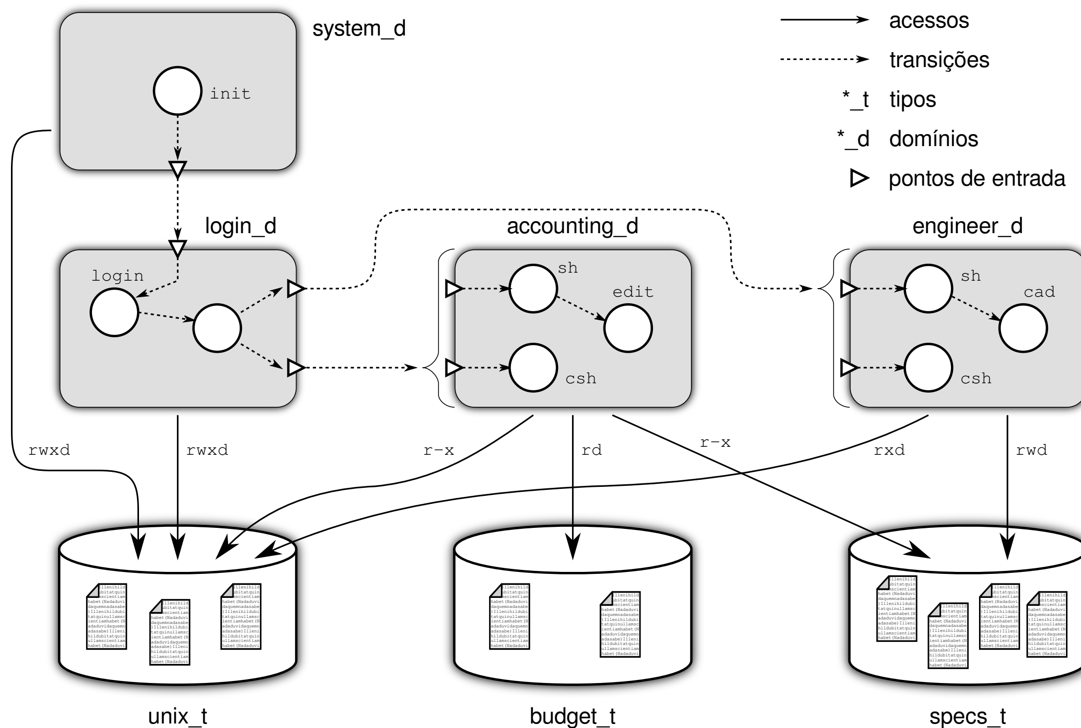

Figura 14: Exemplo de política baseada em domínios e tipos.

de *papéis* no sistema, como "diretor", "gerente", "suporte", "programador", etc. e atribui a cada papel um conjunto de autorizações. Essas autorizações podem ser atribuídas aos papéis de forma discricionária ou obrigatória.

Para cada usuário do sistema é definido um conjunto de papéis que este pode assumir. Durante sua sessão no sistema (geralmente no início), o usuário escolhe os papéis que deseja ativar e recebe as autorizações correspondentes, válidas até este desativar os papéis correspondentes ou encerrar sua sessão. Assim, um usuário autorizado pode ativar os papéis de "professor" ou de "aluno" dependendo do que deseja fazer no sistema.

Os papéis permitem desacoplar os usuários das permissões. Por isso, um conjunto de papéis definido adequadamente é bastante estável, restando à gerência apenas atribuir a cada usuário os papéis a que este tem direito. A Figura 15 apresenta os principais componentes de uma política RBAC.

Existem vários modelos para a implementação de políticas baseadas em papéis, como os apresentados em [Sandhu et al., 1996]. Por exemplo, no modelo *RBAC hierárquico* os papéis são classificados em uma hierarquia, na qual os papéis superiores herdam as permissões dos papéis inferiores. No modelo *RBAC com restrições* é possível definir restrições à ativação de papéis, como o número máximo de usuários que podem ativar um determinado papel simultaneamente ou especificar que dois papéis são conflitantes e não podem ser ativados pelo mesmo usuário simultaneamente.

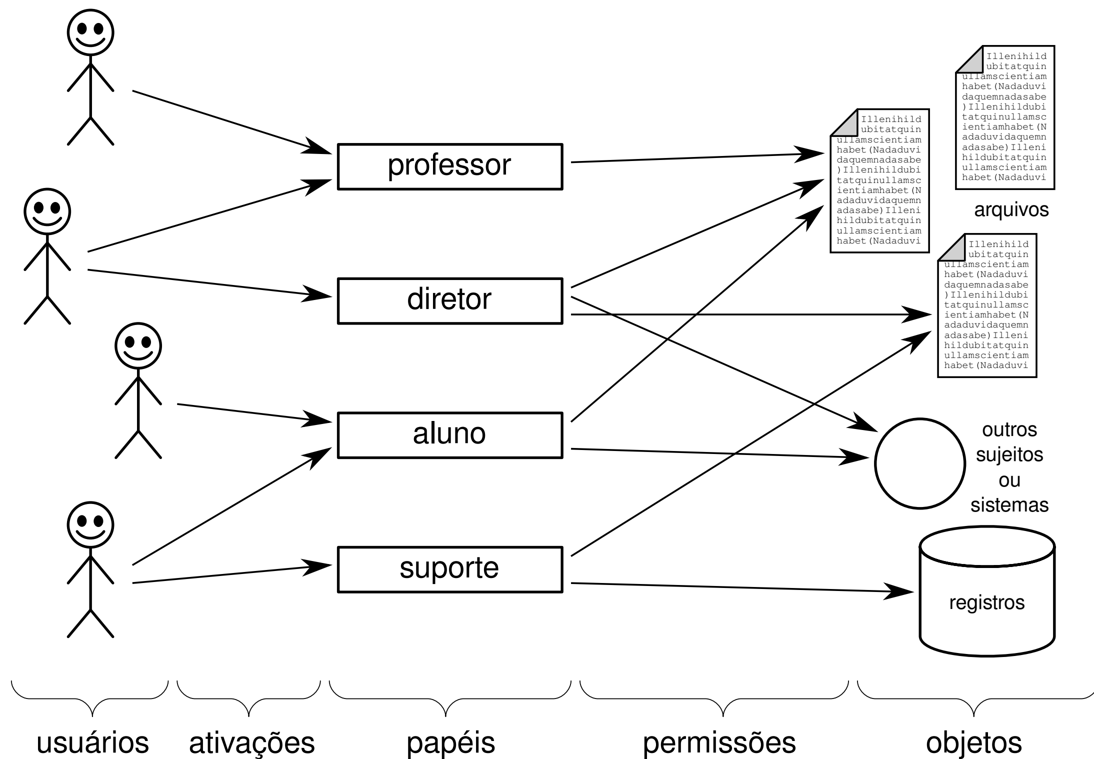

Figura 15: Políticas baseadas em papéis.

5.6 Mecanismos de controle de acesso

A implementação do controle de acesso em um sistema computacional deve ser independente das políticas de controle de acesso adotadas. Como nas demais áreas de um sistema operacional, a separação entre mecanismo e política é importante, por possibilitar trocar a política de controle de acesso sem ter de modificar a implementação do sistema. A infra-estrutura de controle de acesso deve ser ao mesmo tempo *inviolável* (impossível de adulterar ou enganar) e *incontornável* (todos os acessos aos recursos do sistema devem passar por ela).

5.6.1 Infra-estrutura básica

A arquitetura básica de uma infra-estrutura de controle de acesso típica é composta pelos seguintes elementos (Figura 16):

Bases de sujeitos e objetos (*User/Object Bases*): relação dos sujeitos e objetos que compõem o sistema, com seus respectivos atributos;

Base de políticas (*Policy Base*): base de dados contendo as regras que definem como e quando os objetos podem ser acessados pelos sujeitos, ou como/quando os sujeitos podem interagir entre si;

Monitor de referências (*Reference monitor*): elemento que julga a pertinência de cada pedido de acesso. Com base em atributos do sujeito e do objeto (como suas

respectivas identidades), nas regras da base de políticas e possivelmente em informações externas (como horário, carga do sistema, etc.), o monitor decide se um acesso deve ser permitido ou negado;

Mediador (impositor ou *Enforcer*): elemento que medeia a interação entre sujeitos e objetos; a cada pedido de acesso a um objeto, o mediador consulta o monitor de referências e permite/nega o acesso, conforme a decisão deste último.

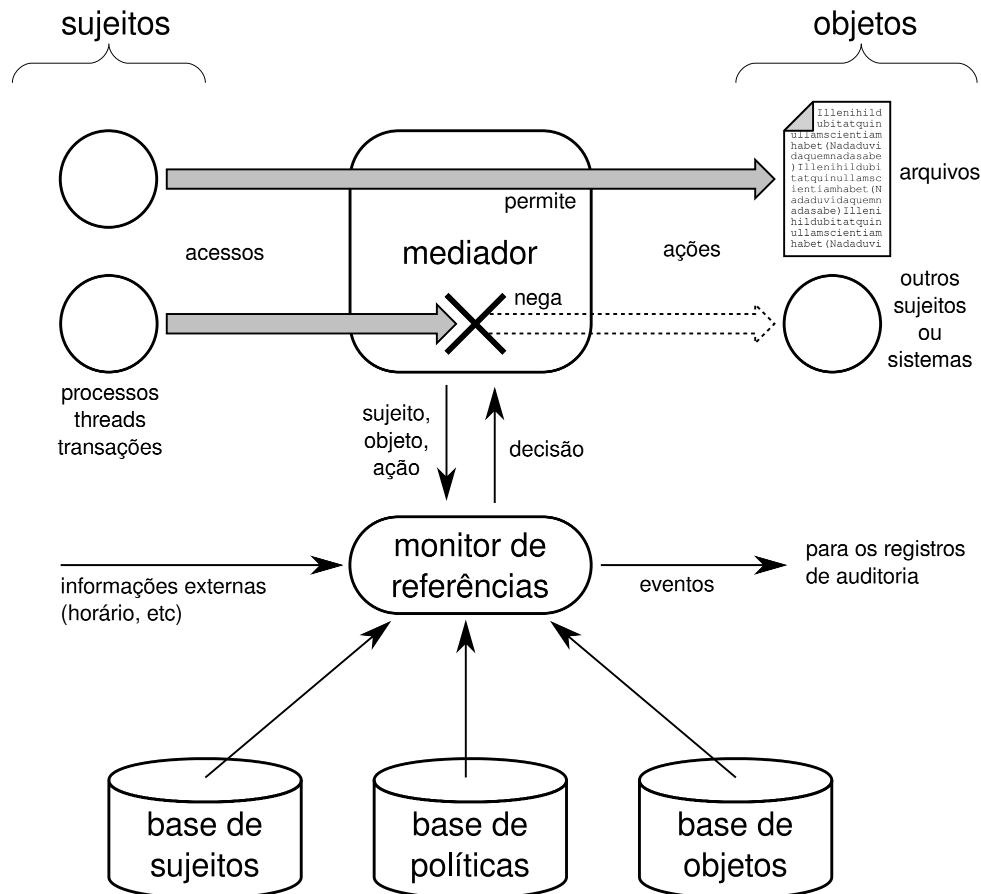

Figura 16: Estrutura genérica de uma infra-estrutura de controle de acesso.

É importante observar que os elementos dessa estrutura são componentes lógicos, que não impõem uma forma de implementação rígida. Por exemplo, em um sistema operacional convencional, o sistema de arquivos possui sua própria estrutura de controle de acesso, com permissões de acesso armazenadas nos próprios arquivos, e um pequeno monitor/mediador associado a algumas chamadas de sistema, como `open` e `mmap`. Outros recursos (como áreas de memória ou semáforos) possuem suas próprias regras e estruturas de controle de acesso, organizadas de forma diversa.

5.6.2 Controle de acesso em UNIX

Os sistemas operacionais do mundo UNIX implementam um sistema de ACLs básico bastante rudimentar, no qual existem apenas três sujeitos: *user* (o dono do recurso),

group (um grupo de usuários ao qual o recurso está associado) e *others* (todos os demais usuários do sistema). Para cada objeto existem três possibilidades de acesso: *read*, *write* e *execute*, cuja semântica depende do tipo de objeto (arquivo, diretório, *socket* de rede, área de memória compartilhada, etc.). Dessa forma, são necessários apenas 9 bits por arquivo para definir suas permissões básicas de acesso.

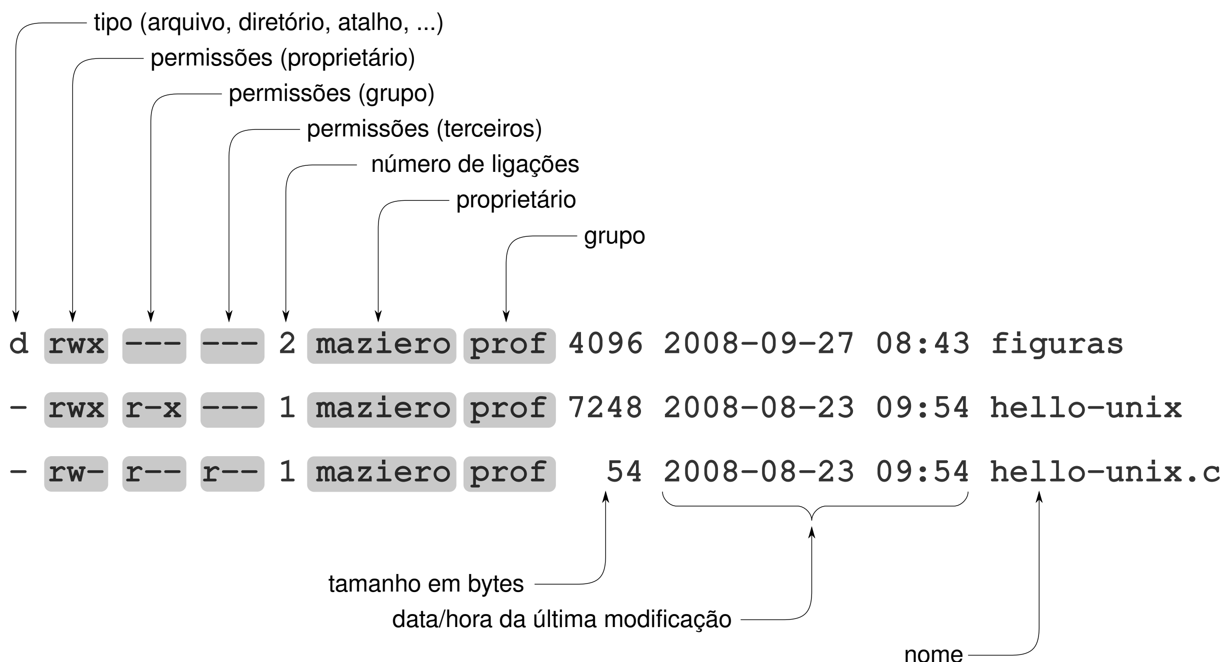

Figura 17: Listas de controle de acesso em UNIX.

A Figura 17 apresenta uma listagem de diretório típica em UNIX. Nessa listagem, o arquivo `hello-unix.c` pode ser acessado em leitura e escrita por seu proprietário (o usuário `maziero`, com permissões `rw-`), em leitura pelos usuários do grupo `prof` (permissões `r--`) e em leitura pelos demais usuários do sistema (permissões `r--`). Já o arquivo `hello-unix` pode ser acessado em leitura, escrita e execução por seu proprietário (permissões `rwx`), em leitura e execução pelos usuários do grupo `prof` (permissões `r-x`) e não pode ser acessado pelos demais usuários (permissões `---`). No caso de diretórios, a permissão de leitura autoriza a listagem do diretório, a permissão de escrita autoriza sua modificação (criação, remoção ou renomeação de arquivos ou sub-diretórios) e a permissão de execução autoriza usar aquele diretório como diretório de trabalho ou parte de um caminho.

É importante destacar que o controle de acesso é normalmente realizado apenas durante a abertura do arquivo, para a criação de seu descritor em memória. Isso significa que, uma vez aberto um arquivo por um processo, este terá acesso ao arquivo enquanto o mantiver aberto, mesmo que as permissões do arquivo sejam modificadas para impedir esse acesso. O controle contínuo de acesso a arquivos é pouco frequentemente implementado em sistemas operacionais, porque verificar as permissões de acesso a cada operação de leitura ou escrita teria um forte impacto negativo sobre o desempenho do sistema.

Dessa forma, um descritor de arquivo aberto pode ser visto como uma capacidade (vide Seção 5.2.4), pois a posse do descritor permite ao processo acessar o arquivo referenciado por ele. O processo recebe esse descritor ao abrir o arquivo e deve apresentá-lo a cada acesso subsequente; o descritor pode ser transferido aos processos filhos ou até mesmo a outros processos, outorgando a eles o acesso ao arquivo aberto. A mesma estratégia é usada em *sockets* de rede, semáforos e outros mecanismos de IPC.

O padrão POSIX 1003.1e definiu ACLs mais detalhadas para o sistema de arquivos, que permitem definir permissões para usuários e grupos específicos além do proprietário do arquivo. Esse padrão é parcialmente implementado em vários sistemas operacionais, como o Linux e o FreeBSD. No Linux, os comandos `getfacl` e `setfacl` permitem manipular essas ACLs, como mostra o exemplo a seguir:

```
1  host:~> ll
2  -rw-r--r-- 1 maziero prof 2450791 2009-06-18 10:47 main.pdf
3
4  host:~> getfacl main.pdf
5  # file: main.pdf
6  # owner: maziero
7  # group: maziero
8  user::rw-
9  group::r--
10 other::r--
11
12 host:~> setfacl -m diogo:rw,rafael:rw main.pdf
13
14 host:~> getfacl main.pdf
15 # file: main.pdf
16 # owner: maziero
17 # group: maziero
18 user::rw-
19 user:diogo:rw-
20 user:rafael:rw-
21 group::r--
22 mask::rw-
23 other::r--
```

No exemplo, o comando da linha 12 define permissões de leitura e escrita específicas para os usuários `diogo` e `rafael` sobre o arquivo `main.pdf`. Essas permissões estendidas são visíveis na linha 19 e 20, junto com as permissões UNIX básicas (nas linhas 18, 21 e 23).

5.6.3 Controle de acesso em Windows

Os sistemas Windows baseados no núcleo NT (NT, 2000, XP, Vista e sucessores) implementam mecanismos de controle de acesso bastante sofisticados [Brown, 2000, Russinovich and Solomon, 2004]. Em um sistema Windows, cada sujeito (computador, usuário, grupo ou domínio) é unicamente identificado por um *identificador de segurança*

(SID - *Security IDentifier*). Cada sujeito do sistema está associado a um *token de acesso*, criado no momento em que o respectivo usuário ou sistema externo se autentica no sistema. A autenticação e o início da sessão do usuário são gerenciados pelo LSASS (*Local Security Authority Subsystem*), que cria os processos iniciais e os associa ao *token* de acesso criado para aquele usuário. Esse *token* normalmente é herdado pelos processos filhos, até o encerramento da sessão do usuário. Ele contém o identificador do usuário (SID), dos grupos aos quais ele pertence, privilégios a ele associados e outras informações. Privilégios são permissões para realizar operações genéricas, que não dependem de um recurso específico, como reiniciar o computador, carregar um *driver* ou depurar um processo.

Por outro lado, cada objeto do sistema está associado a um *descritor de segurança* (SD - *Security Descriptor*). Como objetos, são considerados arquivos e diretórios, processos, impressoras, serviços e chaves de registros, por exemplo. Um descritor de segurança indica o proprietário e o grupo primário do objeto, uma lista de controle de acesso de sistema (SACL - *System ACL*), uma lista de controle de acesso discricionária (DACL - *Discretionary ACL*) e algumas informações de controle.

A DACL contém uma lista de regras de controle de acesso ao objeto, na forma de ACEs (*Access Control Entries*). Cada ACE contém um identificador de usuário ou grupo, um modo de autorização (positiva ou negativa), um conjunto de permissões (ler, escrever, executar, remover, etc.), sob a forma de um mapa de bits. Quando um sujeito solicita acesso a um recurso, o SRM (*Security Reference Monitor*) compara o *token* de acesso do sujeito com as entradas da DACL do objeto, para permitir ou negar o acesso. Como sujeitos podem pertencer a mais de um grupo e as ACEs podem ser positivas ou negativas, podem ocorrer conflitos entre as ACEs. Por isso, um mecanismo de resolução de conflitos é acionado a cada acesso solicitado ao objeto.

A SACL define que tipo de operações sobre o objeto devem ser registradas pelo sistema, sendo usada basicamente para para fins de auditoria (Seção 6). A estrutura das ACEs de auditoria é similar à das ACEs da DACL, embora defina quais ações sobre o objeto em questão devem ser registradas para quais sujeitos. A Figura 18 ilustra alguns dos componentes da estrutura de controle de acesso dos sistemas Windows.

5.6.4 Outros mecanismos

As políticas de segurança básicas utilizadas na maioria dos sistemas operacionais são discricionárias, baseadas nas identidades dos usuários e em listas de controle de acesso. Entretanto, políticas de segurança mais sofisticadas vêm sendo gradualmente agregadas aos sistemas operacionais mais complexos, visando aumentar sua segurança. Algumas iniciativas dignas de nota são apresentadas a seguir:

- O SELinux é um mecanismo de controle de acesso multi-políticas, desenvolvido pela NSA (*National Security Agency*, USA) [Loscocco and Smalley, 2001] a partir da arquitetura flexível de segurança *Flask* (*Flux Advanced Security Kernel*) [Spencer et al., 1999]. Ele constitui uma infra-estrutura complexa de segurança para o núcleo Linux, capaz de aplicar diversos tipos de políticas obrigatórias aos recursos do sistema operacional. A política default do SELinux é baseada em RBAC e DTE, mas ele também é capaz de implementar políticas de segurança

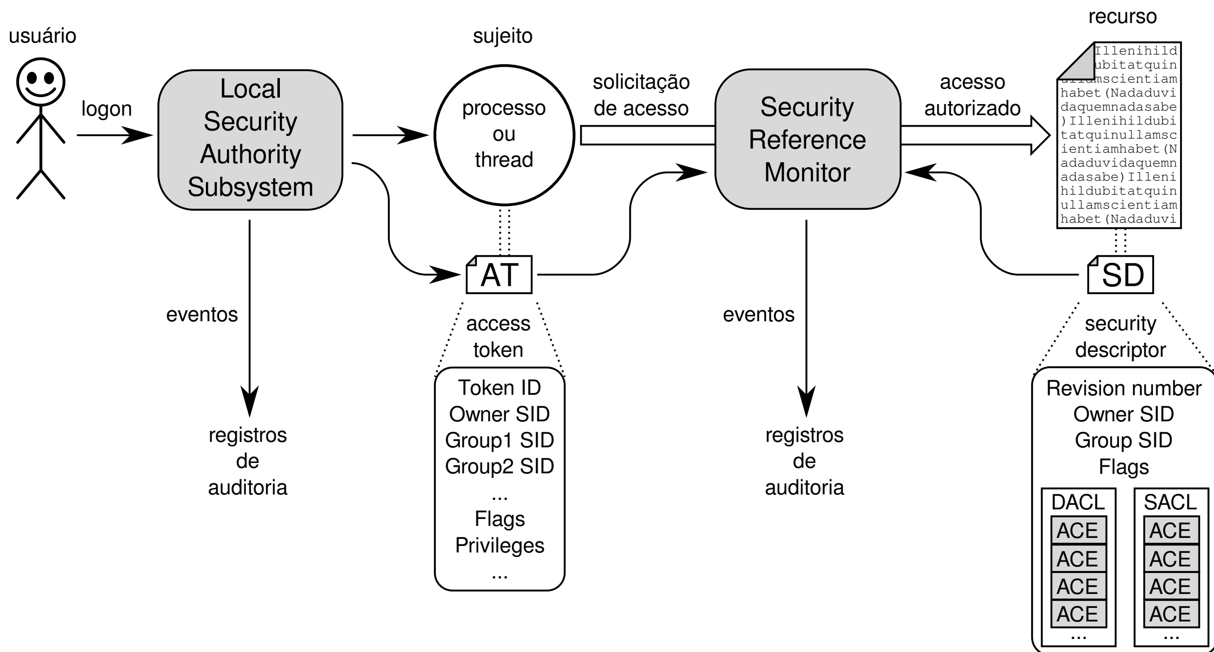

Figura 18: Listas de controle de acesso no Windows.

multi-nível. O SELinux tem sido criticado devido à sua complexidade, que torna difícil sua compreensão e configuração. Em consequência, outros projetos visando adicionar políticas MAC mais simples e fáceis de usar ao núcleo Linux têm sido propostos, como *LIDS*, *SMACK* e *AppArmor*.

- O sistema operacional Windows Vista incorpora uma política denominada *Mandatory Integrity Control* (MIC) que associa aos processos e recursos os níveis de integridade *Low*, *Medium*, *High* e *System* [Microsoft, 2007], de forma similar ao modelo de Biba (Seção 5.3.2). Os processos normais dos usuários são classificados como de integridade média, enquanto o navegador Web e executáveis provindos da Internet são classificados como de integridade baixa. Além disso, o Vista conta com o UAC (*User Account Control*) que aplica uma política baseada em RBAC: um usuário com direitos administrativos inicia sua sessão como usuário normal, e somente ativa seu papel administrativo quando necessita efetuar uma ação administrativa.

- O projeto TrustedBSD [Watson, 2001] implementa ACLs no padrão POSIX, capacidades POSIX e o suporte a políticas obrigatórias como Bell LaPadula, Biba, categorias e TE/DTE. Uma versão deste projeto foi portada para o MacOS X, sendo denominada *MacOS X MAC Framework*.

- Desenvolvido nos anos 90, o sistema operacional experimental *EROS* (*Extremely Reliable Operating System*) [Shapiro and Hardy, 2002] implementou um modelo de controle de acesso totalmente baseado em capacidades. Nesse modelo, todas as interfaces dos componentes do sistema só são acessíveis através de capacidades, que são usadas para nomear as interfaces e para controlar seu acesso. O sistema

EROS deriva de desenvolvimentos anteriores feitos no sistema operacional KeyKOS para a plataforma S/370 [Bomberger et al., 1992].

- Em 2009, o sistema operacional experimental *SeL4*, que estende o sistema micro-núcleo L4 [Liedtke, 1996] com um modelo de controle de acesso baseado em capacidades similar ao utilizado no sistema EROS, tornou-se o primeiro sistema operacional cuja segurança foi formalmente verificada [Klein et al., 2009]. A verificação formal é uma técnica de engenharia de software que permite demonstrar matematicamente que a implementação do sistema corresponde à sua especificação, e que a especificação está completa e sem erros.
- O sistema *Trusted Solaris* [Sun Microsystems, 2000] implementa várias políticas de segurança: em MLS (*Multi-Level Security*), níveis de segurança são associados aos recursos do sistema e aos usuários. Além disso, a noção de domínios é implementada através de "compartimentos": um recurso associado a um determinado compartimento só pode ser acessado por sujeitos no mesmo compartimento. Para limitar o poder do super-usuário, é usada uma política de tipo RBAC, que divide a administração do sistema em vários papéis de podem ser atribuídos a usuários distintos.

5.7 Mudança de privilégios

Normalmente, os processos em um sistema operacional são sujeitos que representam o usuário que os lançou. Quando um novo processo é criado, ele herda as credenciais de seu processo-pai, ou seja, seus identificadores de usuário e de grupo. Na maioria dos mecanismos de controle de acesso usados em sistemas operacionais, as permissões são atribuídas aos processos em função de suas credenciais. Com isso, normalmente cada novo processo herda as mesmas permissões de seu processo-pai, pois possui as mesmas credenciais dele.

O uso de privilégios fixos é adequado para o uso normal do sistema, pois os processos de cada usuário só devem ter acesso aos recursos autorizados para esse usuário. Entretanto, em algumas situações esse mecanismo se mostra inadequado. Por exemplo, caso um usuário precise executar uma tarefa administrativa, como instalar um novo programa, modificar uma configuração de rede ou atualizar sua senha, alguns de seus processos devem possuir permissões para as ações necessárias, como editar arquivos de configuração do sistema. Os sistemas operacionais atuais oferecem diversas abordagens para resolver esse problema:

Usuários administrativos : são associadas permissões administrativas às sessões de trabalho de alguns usuários específicos, permitindo que seus processos possam efetuar tarefas administrativas, como instalar softwares ou mudar configurações. Esta é a abordagem utilizada em alguns sistemas operacionais de amplo uso. Algumas implementações definem vários tipos de usuários administrativos, com diferentes tipos de privilégios, como acessar dispositivos externos, lançar máquinas virtuais, reiniciar o sistema, etc. Embora simples, essa solução é falha, pois se algum programa com conteúdo malicioso for executado por um usuário administrativo, terá acesso a todas as suas permissões.

Permissões temporárias : conceder sob demanda a certos processos do usuário as permissões de que necessitam para realizar ações administrativas; essas permissões podem ser descartadas pelo processo assim que concluir as ações. Essas permissões podem estar associadas a papéis administrativos (Seção 5.5), ativados quando o usuário tiver necessidade deles. Esta é a abordagem usada pela infra-estrutura UAC (*User Access Control*) [Microsoft, 2007], na qual um usuário administrativo inicia sua sessão de trabalho como usuário normal, e somente ativa seu papel administrativo quando necessita efetuar uma ação administrativa, desativando-o imediatamente após a conclusão da ação. A ativação do papel administrativo pode impor um procedimento de reautenticação.

Mudança de credenciais : permitir que certos processos do usuário mudem de identidade, assumindo a identidade de algum usuário com permissões suficientes para realizar a ação desejada; pode ser considerada uma variante da atribuição de permissões temporárias. O exemplo mais conhecido de implementação desta abordagem são os flags `setuid` e `setgid` do UNIX, explicados a seguir.

Monitores : definir processos privilegiados, chamados *monitores* ou *supervisores*, recebem pedidos de ações administrativas dos processos não-privilegiados, através de uma API pré-definida; os pedidos dos processos normais são validados e atendidos. Esta é a abordagem definida como *separação de privilégios* em [Provos et al., 2003], e também é usada na infra-estrutura *PolicyKit*, usada para autorizar tarefas administrativas em ambientes *desktop* Linux.

Um mecanismo amplamente usado para mudança de credenciais consiste dos flags `setuid` e `setgid` dos sistemas UNIX. Se um arquivo executável tiver o flag `setuid` habilitado (indicado pelo caractere "s" em suas permissões de usuário), seus processos assumirão as credenciais do proprietário do arquivo. Portanto, se o proprietário de um arquivo executável for o usuário *root*, os processos lançados a partir dele terão todos os privilégios do usuário *root*, independente de quem o tiver lançado. De forma similar, processos lançados a partir de um arquivo executável com o flag `setgid` habilitado terão as credenciais do grupo associado ao arquivo. A Figura 19 ilustra esse mecanismo: o primeiro caso representa um executável normal (sem esses flags habilitados); um processo filho lançado a partir do executável possui as mesmas credenciais de seu pai. No segundo caso, o executável pertence ao usuário *root* e tem o flag `setuid` habilitado; assim, o processo filho assume a identidade do usuário *root* e, em consequência, suas permissões de acesso. No último caso, o executável pertence ao usuário *root* e tem o flag `setgid` habilitado; assim, o processo filho pertencerá ao grupo *mail*.

Os flags `setuid` e `setgid` são muito utilizados em programas administrativos no UNIX, como troca de senha e agendamento de tarefas, sempre que for necessário efetuar uma operação inacessível a usuários normais, como modificar o arquivo de senhas. Todavia, esse mecanismo pode ser perigoso, pois o processo filho recebe todos os privilégios do proprietário do arquivo, o que contraria o princípio do privilégio mínimo. Por exemplo, o programa `passwd` deveria somente receber a autorização para modificar o arquivo de senhas (`/etc/passwd`) e nada mais, pois o super-usuário (*root user*) tem acesso a todos os recursos do sistema e pode efetuar todas as operações que desejar.

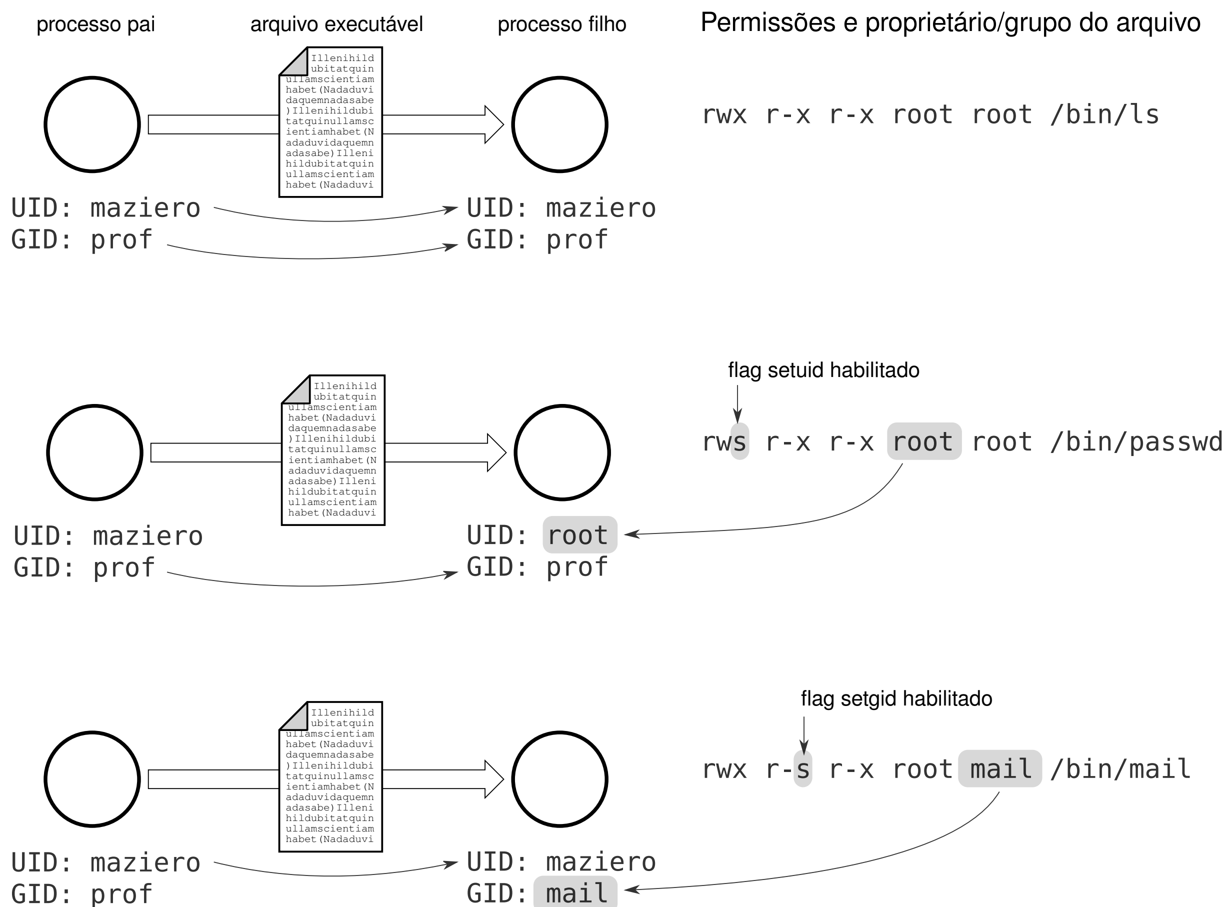

Figura 19: Funcionamento dos flags `setuid` e `setgid` do UNIX.

Se o programa `passwd` contiver erros de programação, ele pode ser induzido pelo seu usuário a efetuar ações não-previstas, visando comprometer a segurança do sistema (vide Seção 2.3).

Uma alternativa mais segura aos flags `setuid` e `setgid` são os *privilégios POSIX* (*POSIX Capabilities*[7]), definidos no padrão POSIX 1003.1e [Gallmeister, 1994]. Nessa abordagem, o "poder absoluto" do super-usuário é dividido em um grande número de pequenos privilégios específicos, que podem ser atribuídos a certos processos do sistema. Como medida adicional de proteção, cada processo pode ativar/desativar os privilégios que possui em função de sua necessidade. Vários sistemas UNIX implementam privilégios POSIX, como é o caso do Linux, que implementa:

- `CAP_CHOWN`: alterar o proprietário de um arquivo qualquer;
- `CAP_USER_DEV`: abrir dispositivos;
- `CAP_USER_FIFO`: usar *pipes* (comunicação);

[7]O padrão POSIX usou indevidamente o termo "capacidade" para definir o que na verdade são privilégios associados aos processos. O uso indevido do termo *POSIX Capabilities* perdura até hoje em vários sistemas, como é o caso do Linux.

- `CAP_USER_SOCK`: abrir *sockets* de rede;
- `CAP_NET_BIND_SERVICE`: abrir portas de rede com número abaixo de 1024;
- `CAP_NET_RAW`: abrir *sockets* de baixo nível (*raw sockets*);
- `CAP_KILL`: enviar sinais para processos de outros usuários.
- ... (outros +30 privilégios)

Para cada processo são definidos três conjuntos de privilégios: *Permitidos* (P), *Efetivos* (E) e *Herdáveis* (H). Os privilégios permitidos são aqueles que o processo pode ativar quando desejar, enquanto os efetivos são aqueles ativados no momento (respeitando-se $E \subseteq P$). O conjunto de privilégios herdáveis H é usado no cálculo dos privilégios transmitidos aos processos filhos. Os privilégios POSIX também podem ser atribuídos a programas executáveis em disco, substituindo os tradicionais (e perigosos) flags `setuid` e `setgid`. Assim, quando um executável for lançado, o novo processo recebe um conjunto de privilégios calculado a partir dos privilégios atribuídos ao arquivo executável e aqueles herdados do processo-pai que o criou [Bovet and Cesati, 2005].

Um caso especial de mudança de credenciais ocorre em algumas circunstâncias, quando é necessário **reduzir** as permissões de um processo. Por exemplo, o processo responsável pela autenticação de usuários em um sistema operacional deve criar novos processos para iniciar a sessão de trabalho de cada usuário. O processo autenticador geralmente executa com privilégios elevados, para poder acessar a bases de dados de autenticação dos usuários, enquanto os novos processos devem receber as credenciais do usuário autenticado, que normalmente tem menos privilégios. Em UNIX, um processo pode solicitar a mudança de suas credenciais através da chamada de sistema `setuid()`, entre outras. Em Windows, o mecanismo conhecido como *impersonation* permite a um processo ou *thread* abandonar temporariamente seu *token* de acesso e assumir outro, para realizar uma tarefa em nome do sujeito correspondente [Russinovich and Solomon, 2004].

6 Auditoria

Na área de segurança de sistemas, o termo "auditar" significa recolher dados sobre o funcionamento de um sistema ou aplicação e analisá-los para descobrir vulnerabilidades ou violações de segurança, ou para examinar violações já constatadas, buscando suas causas e possíveis consequências[8] [Sandhu and Samarati, 1996]. Os dois pontos-chave da auditoria são portanto a *coleta* de dados e a *análise* desses dados, que serão discutidas a seguir.

6.1 Coleta de dados

Um sistema computacional em funcionamento processa uma grande quantidade de eventos. Destes, alguns podem ser de importância para a segurança do sistema,

[8] A análise de violações já ocorridas é comumente conhecida como *análise post-mortem*.

como a autenticação de um usuário (ou uma tentativa malsucedida de autenticação), uma mudança de credenciais, o lançamento ou encerramento de um serviço, etc. Os dados desses eventos devem ser coletados a partir de suas fontes e registrados de forma adequada para a análise e arquivamento.

Dependendo da natureza do evento, a coleta de seus dados pode ser feita no nível da aplicação, de sub-sistema ou do núcleo do sistema operacional:

Aplicação : eventos internos à aplicação, cuja semântica é específica ao seu contexto. Por exemplo, as ações realizadas por um servidor HTTP, como páginas fornecidas, páginas não encontradas, erros de autenticação, pedidos de operações não suportadas, etc. Normalmente esses eventos são registrados pela própria aplicação, muitas vezes usando formatos próprios para os dados.

Sub-sistema : eventos não específicos a uma aplicação, mas que ocorrem no espaço de usuário do sistema operacional. Exemplos desses eventos são a autenticação de usuários (ou erros de autenticação), lançamento ou encerramento de serviços do sistema, atualizações de softwares ou de bibliotecas, criação ou remoção de usuários, etc. O registro desses eventos normalmente fica a cargo dos processos ou bibliotecas responsáveis pelos respectivos sub-sistemas.

Núcleo : eventos que ocorrem dentro do núcleo do sistema, sendo inacessíveis aos processos. É o caso dos eventos envolvendo o hardware, como a detecção de erros ou mudança de configurações, e de outros eventos internos do núcleo, como a criação de *sockets* de rede, semáforos, área de memória compartilhada, reinicialização do sistema, etc.

Um aspecto importante da coleta de dados para auditoria é sua forma de representação. A abordagem mais antiga e comum, amplamente disseminada, é o uso de arquivos de registro (*logfiles*). Um arquivo de registro contém uma sequência cronológica de descrições textuais de eventos associados a uma fonte de dados, geralmente uma linha por evento. Um exemplo clássico dessa abordagem são os arquivos de registro do sistema UNIX; a listagem a seguir apresenta um trecho do conteúdo do arquivo `/var/log/security`, geralmente usado para reportar eventos associados à autenticação de usuários:

```
...
Sep  8 23:02:09 espec sudo: e89602174 : user NOT in sudoers ; TTY=pts/1 ; USER=root ; COMMAND=/bin/su
Sep  8 23:19:57 espec userhelper[20480]: running '/sbin/halt' with user_u:system_r:hotplug_t context
Sep  8 23:34:14 espec sshd[6302]: pam_unix(sshd:auth): failure; rhost=210.210.102.173  user=root
Sep  8 23:57:16 espec sshd[6302]: Failed password for root from 210.103.210.173 port 14938 ssh2
Sep  8 00:08:16 espec sshd[6303]: Received disconnect from 210.103.210.173: 11: Bye Bye
Sep  8 00:35:24 espec gdm[9447]: pam_unix(gdm:session): session opened for user rodr by (uid=0)
Sep  8 00:42:19 espec gdm[857]: pam_unix(gdm:session): session closed for user rafael3
Sep  8 00:49:06 espec userhelper[11031]: running '/sbin/halt' with user_u:system_r:hotplug_t context
Sep  8 00:53:40 espec gdm[12199]: pam_unix(gdm:session): session opened for user rafael3 by (uid=0)
Sep  8 00:53:55 espec gdm[12199]: pam_unix(gdm:session): session closed for user rafael3
Sep  8 01:08:43 espec gdm[9447]: pam_unix(gdm:session): session closed for user rodr
Sep  8 01:12:41 espec sshd[14125]: Accepted password for rodr from 189.30.227.212 port 1061 ssh2
Sep  8 01:12:41 espec sshd[14125]: pam_unix(sshd:session): session opened for user rodr by (uid=0)
Sep  8 01:12:41 espec sshd[14127]: subsystem request for sftp
Sep  8 01:38:26 espec sshd[14125]: pam_unix(sshd:session): session closed for user rodr
Sep  8 02:18:29 espec sshd[17048]: Accepted password for e89062004 from 20.0.0.56 port 54233 ssh2
Sep  8 02:18:29 espec sshd[17048]: pam_unix(sshd:session): session opened for user e89062004 by (uid=0)
Sep  8 02:18:29 espec sshd[17048]: pam_unix(sshd:session): session closed for user e89062004
Sep  8 09:06:33 espec sshd[25002]: Postponed publickey for mzr from 159.71.224.62 port 52372 ssh2
Sep  8 06:06:34 espec sshd[25001]: Accepted publickey for mzr from 159.71.224.62 port 52372 ssh2
Sep  8 06:06:34 espec sshd[25001]: pam_unix(sshd:session): session opened for user mzr by (uid=0)
Sep  8 06:06:57 espec su: pam_unix(su-l:session): session opened for user root by mzr(uid=500)
...
```

A infra-estrutura tradicional de registro de eventos dos sistemas UNIX é constituída por um *daemon*[9] chamado `syslogd` (*System Log Daemon*). Esse *daemon* usa um *socket* local e um socket UDP para receber mensagens descrevendo eventos, geradas pelos demais sub-sistemas e aplicações através de uma biblioteca específica. Os eventos são descritos por mensagens de texto e são rotulados por suas fontes em *serviços* (`AUTH`, `KERN`, `MAIL`, etc.) e *níveis* (`INFO`, `WARNING`, `ALERT`, etc.). A partir de seu arquivo de configuração, o processo `syslogd` registra a data de cada evento recebido e decide seu destino: armazenar em um arquivo, enviar a um terminal, avisar o administrador, ativar um programa externo ou enviar o evento a um *daemon* em outro computador são as principais possibilidades. A Figura 20 apresenta os principais componentes dessa arquitetura.

Os sistemas Windows mais recentes usam uma arquitetura similar, embora mais sofisticada do ponto de vista do formato dos dados, pois os eventos são descritos em formato XML (a partir do Windows Vista). O serviço *Windows Event Log* assume o papel de centralizador de eventos, recebendo mensagens de várias fontes, entre elas os componentes do subsistema de segurança (LSASS e SRM, Seção 5.6.3), as aplicações e o próprio núcleo. Conforme visto anteriormente, o componente LSASS gera eventos relativos à autenticação dos usuários, enquanto o SRM registra os acessos a cada objeto de acordo com as regras de auditoria definidas em sua SACL (*System ACLs*). Além disso, aplicações externas podem se registrar junto ao sistema de logs para receber eventos de interesse, através de uma interface de acesso baseada no modelo *publish/subscribe*.

Além dos exemplos aqui apresentados, muitos sistemas operacionais implementam arquiteturas específicas para auditoria, como é o caso do BSM (*Basic Security Module*) do sistema Solaris e sua implementação OpenBSM para o sistema operacional OpenBSD. O sistema MacOS X também provê uma infra-estrutura de auditoria, na

[9]Processo que executa em segundo plano, sem estar associado a uma interface com o usuário, como um terminal ou janela.

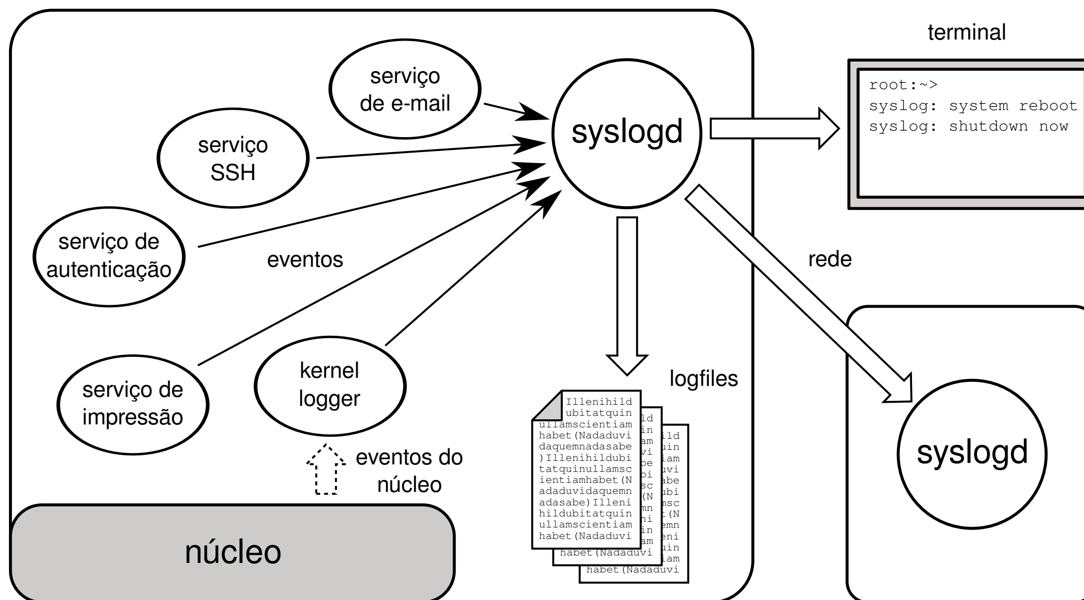

Figura 20: O serviço de *logs* em UNIX.

qual o administrador pode registrar os eventos de seu interesse e habilitar a geração de registros.

Além da coleta de eventos do sistema à medida em que eles ocorrem, outras formas de coleta de dados para auditoria são frequentes. Por exemplo, ferramentas de segurança podem vasculhar o sistema de arquivos em busca de arquivos com conteúdo malicioso, ou varrer as portas de rede para procurar serviços suspeitos.

6.2 Análise de dados

Uma vez registrada a ocorrência de um evento de interesse para a segurança do sistema, deve-se proceder à sua análise. O objetivo dessa análise é sobretudo identificar possíveis violações da segurança em andamento ou já ocorridas. Essa análise pode ser feita sobre os registros dos eventos à medida em que são gerados (chamada análise *online*) ou sobre registros previamente armazenados (análise *offline*). A análise *online* visa detectar problemas de segurança com rapidez, para evitar que comprometam o sistema. Como essa análise deve ser feita simultaneamente ao funcionamento do sistema, é importante que seja rápida e leve, para não prejudicar o desempenho do sistema nem interferir nas operações em andamento. Um exemplo típico de análise online são os anti-vírus, que analisam os arquivos à medida em que estes são acessados pelos usuários.

Por sua vez, a análise *offline* é realizada com dados previamente coletados, possivelmente de vários sistemas. Como não tem compromisso com uma resposta imediata, pode ser mais profunda e detalhada, permitindo o uso de técnicas de mineração de dados para buscar correlações entre os registros, que possam levar à descoberta de problemas de segurança mais sutis. A análise offline é usada em sistemas de detecção de intrusão, por exemplo, para analisar a história do comportamento de cada usuário.

Além disso, é frequentemente usada em sistemas de informação bancários, para se analisar o padrão de uso dos cartões de débito e crédito dos correntista e identificar fraudes.

As ferramentas de análise de registros de segurança podem adotar basicamente duas abordagens: análise por assinaturas ou análise por anomalias. Na *análise por assinaturas*, a ferramenta tem acesso a uma base de dados contendo informações sobre os problemas de segurança conhecidos que deve procurar. Se algum evento ou registro se encaixar nos padrões descritos nessa base, ele é considerado uma violação de segurança. Um exemplo clássico dessa abordagem são os programas anti-vírus: um anti-vírus típico varre o sistema de arquivos em busca de conteúdos maliciosos. O conteúdo de cada arquivo é verificado junto a uma *base de assinaturas*, que contém descrições detalhadas dos vírus conhecidos pelo software; se o conteúdo de um arquivo coincidir com uma assinatura da base, aquele arquivo é considerado suspeito. Um problema com essa forma de análise é sua incapacidade de detectar novas ameaças, como vírus desconhecidos, cuja assinatura não esteja na base.

Por outro lado, uma ferramenta de *análise por anomalias* conta com uma base de dados descrevendo o que se espera como comportamento ou conteúdo normal do sistema. Eventos ou registros que não se encaixarem nesses padrões de normalidade são considerados como violações potenciais da segurança, sendo reportados ao administrador do sistema. A análise por anomalias, também chamada de análise baseada em heurísticas, é utilizada em certos tipos de anti-vírus e sistemas de detecção de intrusão, para detectar vírus ou ataques ainda desconhecidos. Também é muito usada em sistemas de informação bancários, para detectar fraudes envolvendo o uso das contas e cartões bancários. O maior problema com esta técnica é caracterizar corretamente o que se espera como comportamento "normal", o que pode ocasionar muitos erros.

6.3 Auditoria preventiva

Além da coleta e análise de dados sobre o funcionamento do sistema, a auditoria pode agir de forma "preventiva", buscando problemas potenciais que possam comprometer a segurança do sistema. Há um grande número de ferramentas de auditoria, que abordam aspectos diversos da segurança do sistema, entre elas [Pfleeger and Pfleeger, 2006]:

- *Vulnerability scanner*: verifica os softwares instalados no sistema e confronta suas versões com uma base de dados de vulnerabilidades conhecidas, para identificar possíveis fragilidades. Pode também investigar as principais configurações do sistema, com o mesmo objetivo. Como ferramentas deste tipo podem ser citadas: *Metasploit*, *Nessus Security Scanner* e SAINT (*System Administrator's Integrated Network Tool*).

- *Port scanner*: analisa as portas de rede abertas em um computador remoto, buscando identificar os serviços de rede oferecidos pela máquina, as versões do softwares que atendem esses serviços e a identificação do próprio sistema operacional subjacente. O *NMap* é provavelmente o *scanner* de portas mais conhecido atualmente.

- *Password cracker*: conforme visto na Seção 4.3, as senhas dos usuários de um sistema são armazenadas na forma de resumos criptográficos, para aumentar sua

segurança. Um "quebrador de senhas" tem por finalidade tentar descobrir as senhas dos usuários, para avaliar sua robustez. A técnica normalmente usada por estas ferramentas é o ataque do dicionário, que consiste em testar um grande número de palavras conhecidas, suas variantes e combinações, confrontando seus resumos com os resumos das senhas armazenadas. Quebradores de senhas bem conhecidos são o *John the Ripper* para UNIX e *Cain and Abel* para ambientes Windows.

- *Rootkit scanner*: visa detectar a presença de *rootkits* (vide Seção 2.2) em um sistema, normalmente usando uma técnica *offline* baseada em assinaturas. Como os *rootkits* podem comprometer até o núcleo do sistema operacional instalado no computador, normalmente as ferramentas de detecção devem ser aplicadas a partir de outro sistema, carregado a partir de uma mídia externa confiável (CD ou DVD).

- *Verificador de integridade*: a segurança do sistema operacional depende da integridade do núcleo e dos utilitários necessários à administração do sistema. Os verificadores de integridade são programas que analisam periodicamente os principais arquivos do sistema operacional, comparando seu conteúdo com informações previamente coletadas. Para agilizar a verificação de integridade são utilizadas somas de verificação (*checksums*) ou resumos criptográficos como o MD5 e SHA1. Essa verificação de integridade pode se estender a outros objetos do sistema, como a tabela de chamadas de sistema, as portas de rede abertas, os processos de sistema em execução, o cadastro de softwares instalados, etc. Um exemplo clássico de ferramenta de verificação de integridade é o *Tripwire* [Tripwire, 2003], mas existem diversas outras ferramentas mais recentes com propósitos similares.

Referências

[Amoroso, 1994] Amoroso, E. (1994). *Fundamentals of Computer Security Technology*. Prentice Hall PTR.

[Avizienis et al., 2004] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1).

[Badger et al., 1995] Badger, L., Sterne, D., Sherman, D., Walker, K., and Haghighat, S. (1995). Practical Domain and Type Enforcement for UNIX. In *IEEE Symposium on Security and Privacy*, pages 66–77.

[Bell and LaPadula, 1974] Bell, D. E. and LaPadula, L. J. (1974). Secure computer systems. mathematical foundations and model. Technical Report M74-244, MITRE Corporation.

[Biba, 1977] Biba, K. (1977). Integrity considerations for secure computing systems. Technical Report MTR-3153, MITRE Corporation.

[Boebert and Kain, 1985] Boebert, W. and Kain, R. (1985). A practical alternative to hierarchical integrity policies. In *8th National Conference on Computer Security*, pages 18–27.

[Bomberger et al., 1992] Bomberger, A., Frantz, A., Frantz, W., Hardy, A., Hardy, N., Landau, C., and Shapiro, J. (1992). The KeyKOS nanokernel architecture. In *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112.

[Bovet and Cesati, 2005] Bovet, D. and Cesati, M. (2005). *Understanding the Linux Kernel, 3rd edition*. O'Reilly Media, Inc.

[Brown, 2000] Brown, K. (2000). *Programming Windows Security*. Addison-Wesley Professional.

[Cowan et al., 2000] Cowan, C., Beattie, S., Kroah-Hartman, G., Pu, C., Wagle, P., and Gligor, V. (2000). SubDomain: Parsimonious server security. In *14th USENIX Systems Administration Conference*.

[di Vimercati et al., 2007] di Vimercati, S., Foresti, S., Jajodia, S., and Samarati, P. (2007). Access control policies and languages in open environments. In Yu, T. and Jajodia, S., editors, *Secure Data Management in Decentralized Systems*, volume 33 of *Advances in Information Security*, pages 21–58. Springer.

[di Vimercati et al., 2005] di Vimercati, S., Samarati, P., and Jajodia, S. (2005). Policies, Models, and Languages for Access Control. In *Workshop on Databases in Networked Information Systems*, volume LNCS 3433, pages 225–237. Springer-Verlag.

[Gallmeister, 1994] Gallmeister, B. (1994). *POSIX.4: Programming for the Real World*. O'Reilly Media, Inc.

[Jain et al., 2004] Jain, A., Ross, A., and Prabhakar, S. (2004). An Introduction to Biometric Recognition. *IEEE Transactions on Circuits and Systems for Video Technology*, 14(1).

[Klein et al., 2009] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. (2009). SeL4: Formal verification of an OS kernel. In *22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA.

[Lampson, 1971] Lampson, B. (1971). Protection. In *5th Princeton Conference on Information Sciences and Systems*. Reprinted in ACM Operating Systems Rev. 8, 1 (Jan. 1974), pp 18-24.

[Lichtenstein, 1997] Lichtenstein, S. (1997). A review of information security principles. *Computer Audit Update*, 1997(12):9–24.

[Liedtke, 1996] Liedtke, J. (1996). Toward real microkernels. *Communications of the ACM*, 39(9):70–77.

[Loscocco and Smalley, 2001] Loscocco, P. and Smalley, S. (2001). Integrating Flexible Support for Security Policies into the Linux Operating System. In *USENIX Annual Technical Conference*, pages 29–42.

[Menezes et al., 1996] Menezes, A., Van Oorschot, P., and Vanstone, S. (1996). *Handbook of Applied Cryptography*. CRC Press.

[Microsoft, 2007] Microsoft (2007). *Security Enhancements in Windows Vista*. Microsoft Corporation.

[Mitnick and Simon, 2002] Mitnick, K. D. and Simon, W. L. (2002). *The Art of Deception: Controlling the Human Element of Security*. John Wiley & Sons, Inc., New York, NY, USA.

[Mollin, 2000] Mollin, R. A. (2000). *An Introduction to Cryptography*. CRC Press, Inc., Boca Raton, FL, USA.

[Neuman and Ts'o, 1994] Neuman, B. C. and Ts'o, T. (1994). Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38.

[Neuman et al., 2005] Neuman, C., Yu, T., Hartman, S., and Raeburn, K. (2005). The Kerberos Network Authentication Service (V5). RFC 4120 (Proposed Standard). Updated by RFCs 4537, 5021.

[Pfleeger and Pfleeger, 2006] Pfleeger, C. and Pfleeger, S. L. (2006). *Security in Computing, 4th Edition*. Prentice Hall PTR.

[Provos et al., 2003] Provos, N., Friedl, M., and Honeyman, P. (2003). Preventing privilege escalation. In *12th USENIX Security Symposium*.

[Russinovich and Solomon, 2004] Russinovich, M. and Solomon, D. (2004). *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server 2003, Windows XP, and Windows 2000*. Microsoft Press.

[Saltzer and Schroeder, 1975] Saltzer, J. and Schroeder, M. (1975). The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278 – 1308.

[Samarati and De Capitani di Vimercati, 2001] Samarati, P. and De Capitani di Vimercati, S. (2001). Access control: Policies, models, and mechanisms. In Focardi, R. and Gorrieri, R., editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*. Springer-Verlag.

[Sandhu et al., 1996] Sandhu, R., Coyne, E., Feinstein, H., and Youman, C. (1996). Role-based access control models. *IEEE Computer*, 29(2):38–47.

[Sandhu and Samarati, 1996] Sandhu, R. and Samarati, P. (1996). Authentication, access control, and audit. *ACM Computing Surveys*, 28(1).

[Shapiro and Hardy, 2002] Shapiro, J. and Hardy, N. (2002). Eros: a principle-driven operating system from the ground up. *Software, IEEE*, 19(1):26–33.

[Shirey, 2000] Shirey, R. (2000). RFC 2828: Internet security glossary.

[Spencer et al., 1999] Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D., and Lepreau, J. (1999). The Flask security architecture: System support for diverse security policies. In *8th USENIX Security Symposium*, pages 123–139.

[Sun Microsystems, 2000] Sun Microsystems (2000). *Trusted Solaris User's Guide*. Sun Microsystems, Inc.

[Tripwire, 2003] Tripwire (2003). The Tripwire open source project. http://www.tripwire.org.

[Watson, 2001] Watson, R. (2001). TrustedBSD: Adding trusted operating system features to FreeBSD. In *USENIX Technical Conference*.